

N 70 37397

CR 112839

Technical Report 70-113  
NGR 21-002-206

May 1970

A Loader Algorithm for Microprogramming

Oliver Ray Pardo

**CASE FILE  
COPY**



**UNIVERSITY OF MARYLAND  
COMPUTER SCIENCE CENTER  
COLLEGE PARK, MARYLAND**

Technical Report 70-113  
NGR 21-002-206

May 1970

A Loader Algorithm for Microprogramming

Oliver Ray Pardo

This research was supported by Grant NGR 21-002-206 from  
the National Aeronautics and Space Administration to the Computer  
Science Center of the University of Maryland.

### Abstract

This paper describes a loader algorithm for microprogramming for the purpose of performing a unified hardware-software design. The loader is defined here in its broader sense of a relocatable allocator and linkage editor. A loader algorithm is presented in detail in a series of flow charts. To facilitate understanding, the flow charts are explained by an example. Suggestions for future research in implementing the algorithm in microprogram are made.

## TABLE OF CONTENTS

1.0 INTRODUCTION .....	1
2.0 THE DEFINITION OF THE LOADER .....	7
2.1 Terminology	
2.2 Overlay	
2.2.1 The Overlay Structure	
2.2.2 Inter-segment Transfers	
2.2.3 Segmenting	
2.3 The Functions Of A Loader	
2.4 Assumptions And Restrictions	
2.4.1 Assumption On The Machine Environment	
2.4.2 Basic Restrictions On Format	
3.0 INPUTS AND OUTPUT .....	23
3.1 Inputs	
3.1.1 The Relocatable Element As Compiler Output	
3.1.2 Input Specification	
3.1.3 Comparison With Existing Loader Input	
3.2 The Output	
4.0 THE ALGORITHM .....	52
4.1 Structure Of The Data In Memory	
4.2 Detailed Flow Charts	
4.2.1 Auxiliary Routines	
4.2.2 Input And Table Construction	
4.2.3 Global Allocation	
4.2.4 Relocatable Translation And Output	

5.0 THE LOADER IN TERMS OF A UNIFIED HARDWARE-SOFTWARE DESIGN .....	134
5.1 Microprogramming as a Method of Unified Hardware-Software Design	
5.2 Loader Functions Amenable to Microprogram	
5.3 Microprogramming the Translation of Relocatable Code to Executable Code	
6.0 SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS .....	145
APPENDIX .....	149
BIBLIOGRAPHY .....	155

## A Loader Algorithm for Microprogramming

### 1.0 INTRODUCTION

The name "loader" is generally applied to the processor whose input is the output from compilers or assemblers and whose output is an executable code in memory. The earliest loader was a single input command that transmitted machine instructions to memory, while recent loaders have been major systems processors, requiring thousands of instructions to perform a variety of tasks (e.g., the IBM 7090/7094 Loader; the IBM System 360 Linkage Editor; the UNIVAC 1108 EXEC 8 System Collector). The increase in the functional importance of the loader reflects the growth in computer systems.

#### Background

The evolution of the loader from a simple memory allocation routine to its present form is due primarily to the concept of program modularity. Program modularity refers to the partitioning of a program's tasks into subprograms (or subroutines). The idea of subprograms arose early in the development of computers for several reasons:

- (a) Minimize redundant effort. There are many mathematical, data management, and bookkeeping functions that are common to many programs. It is wasteful for each programmer to program a function that has already been programmed.
- (b) Merge subprograms written in different symbolic languages. Most computer systems require that some parts of every program (e.g., Input/Output routines) be written in the machine assembly language. However, it is easier to write large

programs in a higher-order procedural language. Therefore, it is beneficial if the both assembly language and procedure-oriented languages can be used in writing a program.

- (c) Split the programming effort of a large program among several programmers. It is helpful if a program can be designed, compiled and tested in parts and, once completed, assembled into a whole without additional programming.

Initially, subprograms were awkward and hard to use. This was due to the problem of naming conflicts. Name space, the set of addresses generated by a compiler for a program, is distinct from address space, the set of physical memory locations. When a program is located in memory, either the name space must be coincident with the address space, or there must be a fixed procedure to map one space to another.

Obviously, problems occur when the intersection of the address space of two separate subprograms is non-empty. For example, if the name space of subprogram A maps to address space 100 through 150 and the name space of subprogram B maps to address space 120 through 180, a conflict exists in locations 120 through 150. In the earliest computer systems, a subprogram was written for a fixed location in address space. Assembling subprograms into a program required that there were no naming conflicts. This requirement made it extremely difficult to include even a few subprograms from previous applications. It is impossible to imagine procedures of this sort being carried out on modern systems, where the name space of subprograms available to a programmer is several hundred times greater than the address space.

The problem of naming conflicts was resolved by the development of the relocatable subprogram. The idea was to have the compilers and assemblers produce subprograms that

could be located anywhere in memory (address space) and be considered properly located. There are several methods of doing this:

- (1) Addressing relative to the executing instruction . If all addresses are relative to a current pointer, then name space is always coincident with address space.
- (2) Base register addressing . All addressing is relative to the first location in the subprogram. Whatever memory location the first subprogram word occupies is loaded into a base register. In order to make name space and address space coincident, the addresses in the subprogram are added to the base register.
- (3) Adjustment of namespace . This method requires that as a subprogram is loaded into memory, each address be properly modified to coincide with address space.

The third method has been the most common method used in computers although recently both methods (1) and (2) have re-emerged in major systems [1] [2]. Both of these methods require special hardware considerations in the form of registers and logic circuits. This tendency of hardware solutions to follow software solutions will be examined further.

Naming conflicts are not the only problems raised through the introduction of relocatable subprograms. In addition, there are the problems arising from the inherent incompleteness of subprograms:

- How does the compiler resolve undefined references?
- How are the subprograms linked together?
- How are addresses distinguished from data?

The answers to these problems and the resolution of naming conflicts is outside the realm of the compilers and assemblers that compile or assemble subprograms, because the



language processors are not omniscient. In other words, to predict the program that a particular subprogram will belong to is impossible. Indeed, a particular subprogram can become a part of an infinite number of programs.

Therefore, if program modularity is desirable, the role of compilers and assemblers must change. The task of generating executable code must be replaced with the task of generating relocatable subprograms.

Historically, the task of generating an executable program from the set of relocatable subprograms, known as relocatable allocation, was assigned to the loader. For this reason, the name is generally used to cover those processors that perform relocatable allocation. As the loader has evolved, its functions have grown to include many other functions besides relocatable allocation and address linkage. These include subprogram library searching, memory overlay, and, in some systems, file and buffer pool initialization. Recent systems have divorced the actual loading of memory from the program generation allowing the programmer to store the completed program until loading is desired and eliminating the need for re-allocating the set of relocatable subprograms with each request for execution.

#### Purpose

The purpose of this paper is to emphasize the desirability of unified hardware-software design in the area of operating systems while at the same time outlining the functions and method of the loader, a basic subsystem of every operating system. Experience has shown that given a hardware configuration, software can be shaped to perform all the necessary tasks required of the system. Similarly, whatever can be performed in software can surely be implemented directly in hardware. Both cases are expensive and wasteful when carried to extremes. However, the burden of software can often be significantly reduced by the simplest of hardware additions (e.g., index registers). For

this reason, it is desirable to view a computer system in light of the functions that are used frequently.

The approach is not new. The necessity for millions of accurate calculations in various applications in the scientific community has led to sophisticated hardware for high-precision floating-point multiplication and divide. The necessity for precise handling of decimal numbers in large accounting programs has led to the development of hardware for decimal arithmetic. The contention of this paper is that in contemporary systems, some of the programs that are used most frequently are the systems processors themselves. Therefore, it is in this area that we should explore for the correct software-hardware mix.

The loader is a good subsystem to examine in this context. Either the loader itself or the functions it performs are common to all operating systems. It performs a non-trivial set of functions at least once upon most programs that are processed by the computer. The inputs and output of the loader can be described precisely, and the algorithm for loading can be specified with a minimum of knowledge about the environment of the rest of the system which it operates.

The specific approach of the paper will be to define a loader, to present its inputs and output, to present an algorithm for a loader, to discuss the algorithm in terms of unified hardware-software design, to present conclusions, and to make some recommendations as to further areas of study.

The definition of the loader is presented in section 2, accompanying the presentation of terminology, assumptions and restrictions. Section 3 presents the input and output descriptions. In addition, a brief discussion of the differences that exist between loader inputs (e.g., packing of information) is included. Section 4 presents the algorithm for the loader together with a series of flow

charts and accompanying description. Sections 2, 3, and 4 include discussions of alternative methods of formatting the information and performing the loader functions.

Section 5 presents the discussion of the loader from the point of view of a unified hardware-software design. Several functions are specified as warranting closer examination. One function, the translation of relocatable code to executable code, was implemented as a microprogram in an earlier report. This study is referenced, a brief comparison is made with the corresponding functions in an existing system, and the effectiveness of this implementation is discussed.

Section 6 contains the conclusions and recommendations for future study. The appendices present a syntactic and semantic specification of the input to the loader in Backus-Naur Form and a bibliography.

This paper is meant to serve two classes of readers. The first set of readers may use this paper to learn some of the basic techniques involved in the execution of a loader. For the other class of readers (those interested in microprogramming) this paper has been organized to point directly to a microprogram implementation of the algorithm presented.

## 2.0 THE DEFINITION OF THE LOADER

In presenting the definition of the loader, we will discuss its evolutionary development, outlining the ontogeny of a program from symbolic code to executable code. In addition, functions of the loader described within this paper will be outlined, and assumptions about and restrictions to the algorithm presented.

### 2.1 Terminology

Most contemporary digital computers are designed to execute programs consisting of machine instructions with absolute addresses. These instructions are stored in a memory consisting of a set of registers, each with a unique memory address. An absolute address differs from a memory address by, at most, a fixed constant. In particular, if  $M$  is the set of memory addresses, and  $b$  is a hardware address, then  $A$  is the set of absolute addresses permissible for a program,  $A = \{a | a = m - b, a \geq 0, \text{ for every } m \text{ in } M\}$ . A program written in such machine instructions is called an executable code. The set of instructions that the machine will execute is called the machine language. For this paper a program can be considered to be a block of  $n$  memory words assigned absolute addresses  $a(0), a(1), \dots, a(n-1)$  (where  $a(i) = a(i-1) + 1$ ) and consisting of machine instructions and data. Within this paper, the requirement that distinguishes a program from a subset of the program is that no machine instruction may directly reference an absolute address

outside the range  $a(0)$  to  $a(n-1)$ .

Programs are written either in a symbolic assembly language or in a symbolic procedural language or both. The symbolic code used in the language refers to items and locations within the program through assigned symbolic addresses. It is usually impossible for a programmer to grasp all of the details of a large program all at once so that programs are rarely written in their entirety as one large interlocked algorithm. In general, they are written as a set of control sections. A control section is a contiguous piece of a program, either a subprogram or part of a subprogram or a common area. A subprogram is a part of a program. Subprograms reference locations both within and without themselves. A reference to another subprogram is called an external reference. A location in a subprogram that is assigned a symbolic name and that can be referenced from another subprogram is called a global address. An address that is an arithmetic combination of addresses within and without the subprogram is called a complex address. In addition, subprograms can specify a data area as common to several subprograms. Such an area is called a common area. If the contents of the common area are specified, it is said to be a defined common area. An example of a defined common area is the FORTRAN block data. If the contents of the common area are not specified in a subprogram, then it is said to be an undefined common area and is considered not to be a part of a subprogram.

The translation from symbolic code to executable code occurs in two steps. During the first step, the symbolic code of a subprogram is translated by an assembler (in the case of assembly language) or a compiler (in the case of a procedural language) to an intermediate form that can be linked with the other subprograms to form the complete program. In this manner, subprograms written in different symbolic languages can be joined together. This

intermediate form is called a relocatable element because the subprogram contains information which allows it to be located at an absolute address. The machine instructions and data in the relocatable element are called a relocatable code. Addresses within the subprogram are relative to the beginning of the subprogram and are called relative addresses. The relocatable element consists of the relocatable code and tables of the symbolic external references, the symbolic global addresses, the symbolic common areas, and the complex addresses. Functions that are often used by one or more programmers are written as subprograms, translated into relocatable elements, and stored on mass storage in a subprogram library for later use.

At this point, the concept of name space and address space are re-introduced [3]. Address space is the set of memory locations and is denoted  $M$ . Name space is the set of addresses used by a program to refer within itself. This includes both internal and external addresses. The set of internal addresses is called the local name space while the set of external addresses is called the global name space. If we denote the local name space by  $N$ , then for a set of control sections  $\{c_1, c_2, c_3, \dots, c_k\}$ , each has its own local name space  $N(c_1), N(c_2), N(c_3), \dots, N(c_k)$ . The local name space generated by a compiler or assembler for a subprogram  $J$  requiring  $n$  locations in memory is the set  $N(J) = \{0, 1, 2, \dots, n-1\}$ . These are the relative addresses defined above.

The second step of the translation to executable code for a set of control sections  $\{c_1, c_2, \dots, c_k\}$  is to adjust the local name space of each subprogram such that  $N(c_i) \cap N(c_j) = \emptyset$ , for  $1 \leq i \leq k, 1 \leq j \leq k, i \neq j$  (where  $\emptyset$  is the null set). This process is called relocatable allocation and consists of assembling all subprograms necessary to complete the program, linking all external references to the

corresponding global addresses, adjusting the size of common areas to reflect the largest declared size, assigning each control section  $C_i$  an absolute address such that  $N(C_i)$  is unique, computing the complex addresses, and translating all relocatable code to executable code. This final step adjusts all relative addresses in each control section to conform to the new name space.

The product of the translation is an executable program, suitable for loading into memory, called an absolute element. The absolute element consists of (a) the executable code, (b) information as to the location at which the executable code is to be loaded, and (c) the starting address. The starting address is the absolute address of the first instruction to be executed.

## 2.2 Overlay

The previous section ignored the problem created by the group of programs for which the name space exceeds the address space. The solution to the problem is to have a mechanism for placing only a portion of a program's name space into address space at any one time. Many specific methods of implementation exist such as paging and overlay. This report will limit itself to overlay.

Overlay applies to those programs that require a sequence of separate functions, represented as one or more subprograms, to be applied to an input data set. The aggregate of all the subprograms exceed the available memory space, but the requirement of memory space at any one time lies well within the limits.

The technique of overlay is outlined by the following simple example. The user program is divided into blocks of contiguous code, termed segments, the functions of which are

mutually exclusive. Each segment contains one or more control sections. The segment containing the starting address is called the main segment. The main segment, always resident in memory, defines the flow of the program, maintains all information that must be transferred between the subprograms that are in non-resident segments, and contains subprograms that are referenced by more than one segment. The remaining memory is defined as the overlay space, and segments are loaded into this area as requested by the main segment, overlaying the segment previously occupying the area. As will be seen below, the user is not required to have a resident segment nor is he limited to a single level of overlay segments.

Responsibility for overlay is split between the user and the system. As the user can most easily recognize the general flow of control in his program, he must supply the necessary segmentation and structural information. The system, in particular the loader, provides the allocation of space, recognizes inter-segment accesses, and provides a mechanism for loading a requested segment. It should be noted that most programs do not require overlay and that the use of overlay is optional to the user.

### 2.2.1 The Overlay Structure

The concept of a main program calling a series of independent subprograms, each subprogram in a separate segment, lends itself to extension. If each subprogram is viewed the same way (i.e., as being a sequence of mutually independent subprograms called by a main program), then the segmented structure can be theoretically extended to an indefinite number of levels.

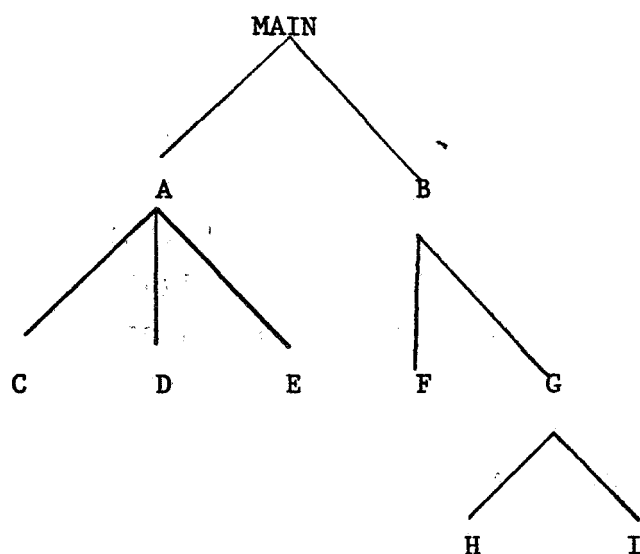


An interpretation of this extension to the overlay structure is the tree structure. In the tree structure, the main segment is termed the root, and all segments it directly references are considered as roots to subtrees. A single segment sub-tree is called a twig. Since, in a tree, there are no circuits, the definition implies that between the root and any twig there is a unique path.

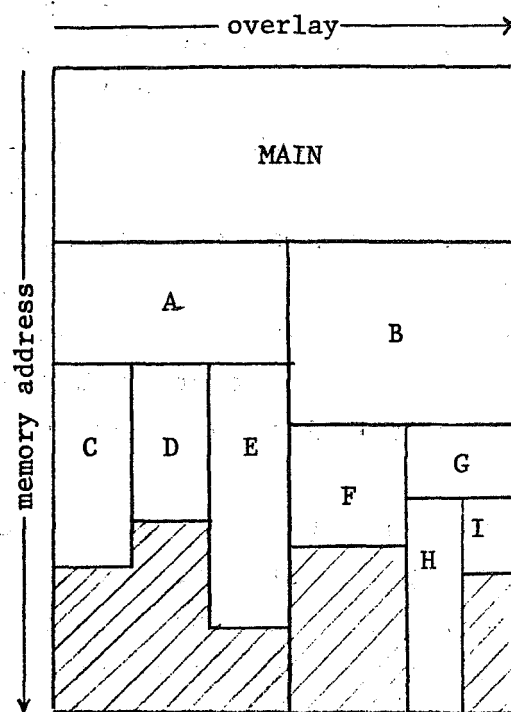
Within the structure, a sub-tree's root segment has a level defined by the number of segments above it (toward the root). For example, the main segment's level is 0, the segments it refers to have level 1, etc. Figure 1(a) shows a tree structure of a segmented program and Figure 1(b) the corresponding memory allocation of the segments. The horizontal lines separate segments that occupy core at the same time; the vertical lines separate segments that overlay each other. For example, branches A-E and B-G-H cannot exist in memory at the same time.

Many conventional loaders use the concept of a tree-structure overlay. This philosophy requires that any one segment can directly access only its own descendants or segments of which it is a descendant. The advantages to this type of structure are in the placement of control sections (common areas or subprograms) referenced in overlaying branches. These are logically placed in a root segment of both branches. Both the IBM 7090/7094 Loader, IBLDR [4] and the IBM System 360 Linkage Editor [5] employ the tree structure. However, the Linkage Editor allows the user to divide his address space into regions, each with its own tree structure, and allows references between regions.

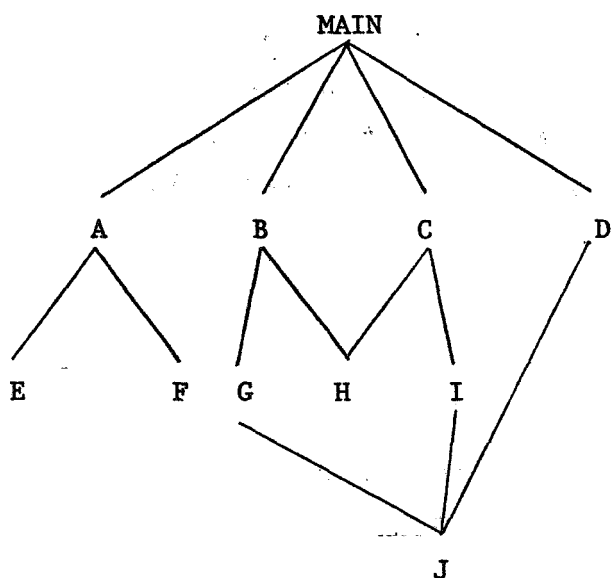
(a) Tree structure



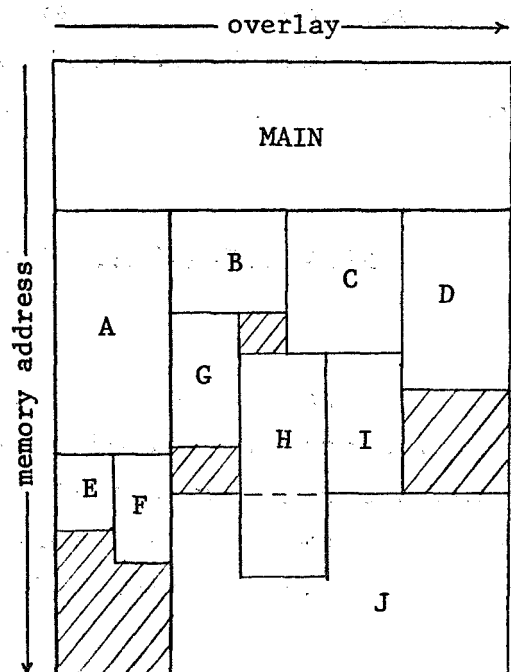
(b) Memory allocation of the tree structure



(c) Non-tree structure



(d) Memory allocation of the non-tree structure




 not used

Figure 1

However, as the responsibility for definition of the overlay structure is given to the user, it seems unwise to impose any structural limitations upon him that can be avoided. In particular, the imposition of a tree structure does not allow separate branches to reference the same subprogram unless that subprogram exists in a segment that is a common root to both. For example, in Figure 1(a), segments D and F can only refer to subprograms in segment MAIN. (Arguments against the rigid tree structure have been outlined by Bernadine Lanzano in [6]).

In order to allow as much freedom as possible, the loader described herein leaves the definition of the structure to the user. This is the method used by the UNIVAC Collector [7]. A multi-leveled structure is available, but there is no implicit tree structure in it. Segments that directly precede a segment in the structure are called the predecessors of that segment, while the segments that directly follow the segment are called the successors of that segment. Figures 1(c) and 1(d) present an example of this type of structure and allocation. In the example, segments A,B,C, and D are successors of segment MAIN, but segments G,I, and D are predecessors of segment J. It should be noted that the names in the example are assigned to the segments and not to memory locations (i.e., the name implies an area of memory, not a point in memory).

### 2.2.2 Inter-segment Transfers

An instruction that transfers control from one segment to another is called an inter-segment transfer. A loading subprogram (named \$LINK\$ herein), is provided by the system to perform inter-segment transfers. Its task is non-trivial. It must identify the instruction request with a

particular location in a segment, check to see if that segment is in memory and if not, load the segment into the proper location in main memory. Once loaded, control must be transferred to the correct location in the segment. There are extensions of this list of requirements; for example, the saving of the block of memory which was overlaid. This paper will not be concerned with further refinements of the overlay concept. Suggestions for future research will be found at the conclusion of this paper.

An inter-segment transfer is possible only from branch instructions referring to a global address. Only these addresses allow the loader to determine the segment to which the transfer is being made. This does not include branch instructions such as subprogram returns (e.g., in most modern computers, these are instructions that branch to the address contained in an index register). In these return instructions there is no way to recognize that the branch instruction should be preceded by a segment overlay. This is the primary reason that inter-segment transfers are to be avoided between segments that overlay each other. An exception to this is when all references between segments that overlay each other are explicit, as in the case of co-routines [13] [14]. In the case of explicit references between segments, the necessity of a resident main program is obviated. Therefore, the only required resident portion of the program is the segment loading subprogram and the linkage table.

The mechanism can be described as follows: (1) for every global address that is referred to from another segment and that is in a non-resident segment (only the main segment is resident), provide a unique resident address called the linkage address, (2) at this address place the actual absolute address of the global address, its corresponding segment identifier, and a branch instruction that branches to a routine, \$LINK\$; (3) at each reference to

the global address from another segment, furnish the corresponding linkage address; and (4) include in the main segment the routine, \$LINK\$, and a list of segments and their mass storage locations. The effect is to force a branch to the dynamic loading routine, \$LINK\$, at every inter-segment transfer. \$LINK\$ loads the correct segment, if not in core, and transfers to the correct address. The definition of the entries at the linkage addresses (linkage table) and the segment list (segment table) can be found in section 2.2.

In the cases in which the user desires to overlay segments that contain only data, a different facility is required. As no transfer will occur, automatic loading is difficult to implement. However, a simple solution is to have the user request a segment be loaded. This is easily effected by adding an entry point in \$LINK\$ that is branched to whenever loading is necessary and adding the segment name to the segment table.

Another enhancement to the system would be to have \$LINK\$ "swap" segments (i.e., copy-out and copy-in) in order to preserve the changes made within any one segment. It should be noted that this requires changes only to the definator of \$LINK\$ and not to the loader algorithm.

### 2.2.3 Segmenting

The segmenting of the program is primarily a responsibility of the user. The user must specify which of his subprograms must appear within each segment. However, it is the loader which performs the subprogram library search and attaches these to the program. In addition, the loader must decide where to locate the undefined common areas. To which segment does the loader attach the library subprogram or undefined common area? The answer must

satisfy three rules:

- (1) The loader should never be responsible for inter-segment transfers between segments that overlay each other. This is a decision that must be left to the user.
- (2) The loader should attempt to minimize the number of segment loads as these require a suspension of computation.
- (3) The loader should not be responsible for inter-segment referencing of data. References of this type do not involve a branch of control and make the task of automatic segment loading extremely difficult. (Note that this does not preclude demand loading of data segments through \$LINK\$.)

The solution chosen to solve the overlay problem is based upon its simplicity of implementation. As the loader encounters references to library subprograms, it attaches the subprogram to the segment that contains the reference. If a library subprogram in segment A references a global address in a segment which will overlay segment A, then the library subprogram is detached from segment A and attached to the main segment (always resident). This also occurs if the library subprogram in segment A is referenced from any other segment. Undefined common areas are attached to the main segment automatically. This solution satisfies the three rules.

Although the above method is simple to implement, it may increase the size of the main segment an undue amount. A more satisfactory method, although more complicated, would be to borrow the method of the tree-structured overlay, and move the common area or subprogram to the nearest common predecessor segment.

### 2.3 The Functions Of A Loader

Within this paper, the functions of the loader will be defined as those functions occurring in the second step of the translation from symbolic code to executable code and the implementation of overlay (sections 2.1 and 2.2). Briefly, these can be outlined as:

- (a) subprogram collection,
- (b) inter-subprogram linkage,
- (c) adjustment of name space,
- (d) relocatable translation,
- (e) overlay implementation,
- (f) loading of the program.

These are primarily the functions performed by the loaders in systems such as the GE 635, CDC 6000 series, IBM 7090/7094 IJOB System and the UNIVAC 1107 EXEC 2 System. However, more recent systems have included processors that perform all the functions short of program loading. Examples of these are the Linkage Editor of the IBM 360 Series and the Collector of the UNIVAC 1108. A few systems have a different concept of name space and require that only a subset of these functions be performed. The Burroughs B5500 uses addressing relative to hardware registers and therefore does not require adjustment of the name space at load-time [1]. The MULTICS system performs its loading dynamically at execute time in a virtual memory system and therefore includes these functions separately within the whole system structure [2].

Therefore, although the functions previously assigned to the loader are now being reassigned, these functions are being performed by modern systems. For this reason, we are justified in studying the loader as a single subsystem.

## 2.4 Assumptions And Restrictions

In addition to the criteria of portability, reliability, and short lead time presented in section 1, the algorithm is designed with the following criteria in mind:

- (a) Simplicity . The algorithm should be easy to follow.
- (b) Efficiency . The algorithm should not waste processing time or memory needlessly.
- (c) Minimum restriction on user . This algorithm should minimize user restrictions.
- (d) Machine-independence . The algorithm should not make undue assumptions as to the hardware configuration or basic operating system philosophy.

The first three criteria are used as the basis for decision-making where alternatives present themselves. The fourth criteria is met by presenting the algorithm in the environment of a minimal system. Indeed, the loader was chosen for this study in that it itself defines a minimal operating system.

### 2.4.1 Assumption On The Machine Environment

The loader described is considered to be one subsystem of the operating system. The operating system is defined as a number of subsystems (language processors, file routines, the I/O control system, etc.) under control of a monitor program. One of the monitor's functions is to interpret system control commands in the input stream. For some input commands, the monitor loads a program from mass storage to memory and relinquishes control to the program. A program



may be a system processor or a user program. Upon completion of the program the monitor assumes control and interprets the next control command.

The operating system operates within a conventional digital computer consisting of a central processing unit that operates on the contents of the main memory. An auxiliary memory, termed mass storage, is provided for information storage. The main memory, hereafter called the memory, consists of a set of addressable registers, the contents of which are called memory words. Mass storage will be considered to be a number of addressable storage devices, called I/O units that store information lineally in blocks of information called records. A record is defined as containing an integral number of memory words. Any one I/O unit is limited to addressing records (i.e., single words cannot be distinguished in the I/O unit).

The algorithm is based in a word addressable memory but it is easily translated to a byte addressable machine. In fact, once the algorithm is examined it will be seen that a byte addressable machine would be preferable.

In order to clarify the algorithm, an example is traced through the major steps. This requires that the machine environment in which the loader operates be specified as to word and field lengths. It should be noted that this is done not to restrict the algorithm, but to clarify the example.

The format chosen is basically that of the IBM 7090/7094 [8]. This has the advantage of allowing a direct comparison of two system approaches. A memory word is defined as 36 bits of information. These bits will be numbered left to right from 0 to 35. Each memory word is addressed by a 15-bit address. There are two address fields in a memory word, bits 3 through 17 in the first half of the word and bits 21-35 in the second half of the word. Each word is logically divided into 6 groups of 6 bits each

called characters and 4 groups of 9 bits each called bytes. This distinction between the character size and byte size is made solely because it is convenient to the description of the algorithm.

#### 2.4.2 Basic Restrictions On Format

As information in a computer is all numerical (coded), a method of interpretation must exist which is non-ambiguous. The interpretation of a piece of coded information is not a function of the data, rather it is a function of the interpreter (e.g., if an arithmetic instruction expects to find a floating-point number at a given address it will interpret whatever data is in that location as a floating point number). Therefore, the location and extent of a piece of information must be made known to the interpreting function.

An addressable piece of information is a field. Fields may have a variety of sizes (the larger the field, the more information it can hold). The location of a field is specified by an address. An address is always relative to some point: the first location in core, another address, or simply the next piece of information. In a binary computer, even though conventional addressing may be word or byte (an integral part of a word), it is generally possible to extract information by bits. A field may be comprised of a single bit (if there are only two conditions for the piece of information it represents).

The address of a field marks the beginning of the field. There are two conventions for indicating the extent of a field. One is to mark the end of a field with some unique code. The other is to provide a priori knowledge of the field length.

The problems with using a mark to denote an end-of-field condition are: (1) a specific code pattern is made

unavailable, and (2) interpretation of fields of this type is slower because of the search and test operations. A priori knowledge of field lengths presents itself as the best alternative in this application for several reasons: (1) the knowledge exists as a by-product of the relocatable machine language generation, (2) efficient use can be made of small fields, and (3) retrieval of a field of information is faster.

### 3.C Inputs And Output

This section describes the inputs and output of the loader in detail. The input to the loader is the output from the system's language processors. As this report covers the function of the loader alone, it is essential that it not change the relationship between the language compilers and the loader. This means that it should not take over any functions of or place any further requirements upon the language processors. Therefore, the input defined below differs little from that of existing loaders in content, although the format may be somewhat unique.

The output from the loader is the absolute element, an executable program stored on mass storage but in a form that allows it to be loaded easily into memory for execution. Two of the reasons behind the decision to separate the collection and linking of relocatable elements from the loading of memory are given below. First, the actual loading of memory is a simple function that has little to do with the formation of executable code and more rightfully belongs as a part of the resident monitor. Second, the output from the loader is an executable program that can be stored indefinitely. This decreases the amount of time required to execute a program as it removes the assembling of relocatable elements. The trade-off is that it creates more stored information. However, mass storage is generally inexpensive compared to processing time.

### 3.1 Inputs

The input to a loader is a set of user-specified subprograms and, possibly, a user-specified overlay structure. The subprograms are in the form of relocatable elements, output from a compiler or assembler. This section presents a discussion of the relocatable element as generated by a compiler, a specification for a loader input (to be used by the algorithm in section 4), and a brief comparison of this input specification with input to existant loaders. A concise description of the input to the loader in Backus-Naur Form is presented as an appendix to this paper.

#### 3.1.1 The Relocatable Element As Compiler Output

The role of a compiler or assembler as a syntax-checker or machine instruction generator is not severely limited by the fact that its input is subprogram (incomplete) rather than a program (complete). However, it is required to do more in the first case.

First, the compiler must specify in some manner which portions of the program will change in the name space adjustment. In most cases, this means each address field. Second, it must preserve a list of those symbolic addresses that remained undefined at the end of compilation (undefined addresses) and a list of their occurrences. Third, it must maintain a list of those locations within it that can be symbolically referenced (global addresses). Fourth, it must maintain a list of data areas that are to be allocated outside the subprogram for common use (common areas).

Finally, it must keep track of the size of the subprogram it has generated.

The result of the effort described above, properly formatted, is the relocatable element. As long as a compiler or assembler retains this information, the formatting is simple. In fact, recent systems (e.g., UNIVAC 1108) include one subprocessor that generates relocatable elements for all its language processors. Therefore, reformatting the relocatable element becomes a simple problem of changing one program.

### 3.1.2 Input Specification

The input to the loader consists of a series of control records. A control record is defined here as a record whose first word is a control word recognized by the system or by a system processor (in this case, the loader). The set of control words that are defined as recognizable are: \$ABSLT, \$SEG, \$RELOC, \$UND, \$DEF, \$COMN, \$CMPLX, and \$TEXT. The name of the control record is defined as the control word.

The control record \$ABSLT, shown in Figure 2(a), causes the monitor program to give control to the loader. It is a symbolic record consisting of the control word \$ABSIT followed by at least one blank space and five fields (ANAME, CUTU, INU, LIB1, CK) separated by commas. The field ANAME is the symbolic name to be assigned to the absolute element by the loader. The field CUTU is an integer specifying the I/C unit on which the absolute element is to be stored. The field INU is an integer specifying the I/C unit on which the user input is stored. The field LIB1 is an integer specifying the I/C unit on which the library index is stored. The field CK is a logical flag (values true or

(a) \$ABSLTb ANAME,OUTU,INU,LIB1,OK

ANAME	the symbolic name to be assigned the absolute element
OUTU	the unit on which the absolute element is to be placed
INU	the unit where the input stream is to be found
LIB1	the unit upon which the library index is to be found
OK	a switch that indicates that processing should or should not proceed if an error condition occurs

(b) \$SEGbbb SNAME (PRED<sub>1</sub>, PRED<sub>2</sub>, . . . ,PRED<sub>n</sub>)

SNAME	the symbolic name of the segment of elements which follows this control card
PRED <sub>i</sub>	the symbolic name of one of n segments which must physically precede this segment in core.

Figure 2 The symbolic control records \$ABSLT and \$SEG

false) that specifies the error action to be taken if a non-fatal error occurs. Any or all of the fields may be omitted. In this case the loader assigns standard values to each.

The control record \$SEG, shown in Figure 2(b), is a symbolic record that is used to define the beginning of a program segment. As mentioned before, a program segment is a set of subprograms and common areas and is defined only when memory overlay is desired. In the case of memory overlay, a segment's location is defined by the location of the segments that must occupy core simultaneously. Therefore, the format of the \$SEG control record is a field SNAME that is the unique symbolic name assigned to the segment and a list of symbolic names (PRED1, PRED2, . . . , PREDN), enclosed in parentheses, of the n segments that may occupy core simultaneously with segment SNAME and that immediately precede the segment SNAME in the overlay structure being defined. This form of overlay description is borrowed from the UNIVAC Collector [7].

As mentioned, the result of assembling an assembly language subprogram or compiling a procedural language subprogram is a relocatable element. A relocatable element is the intermediate form of a subprogram in its translation to executable form. The relocatable element is identified by the control record \$RELOC. This control record, shown in Figure 3(a), consists of three words. The first word is the control word \$REICC, the second word is the symbolic name assigned to the element, and the third word, in two fields, indicates (1) whether or not this program is a main program (if so, this field is also the relative address of the main entry point), and (2) the number of words that this subprogram will require in memory.

The relocatable element consists of two parts, (a) relocatable code and (b) symbolic address tables. The



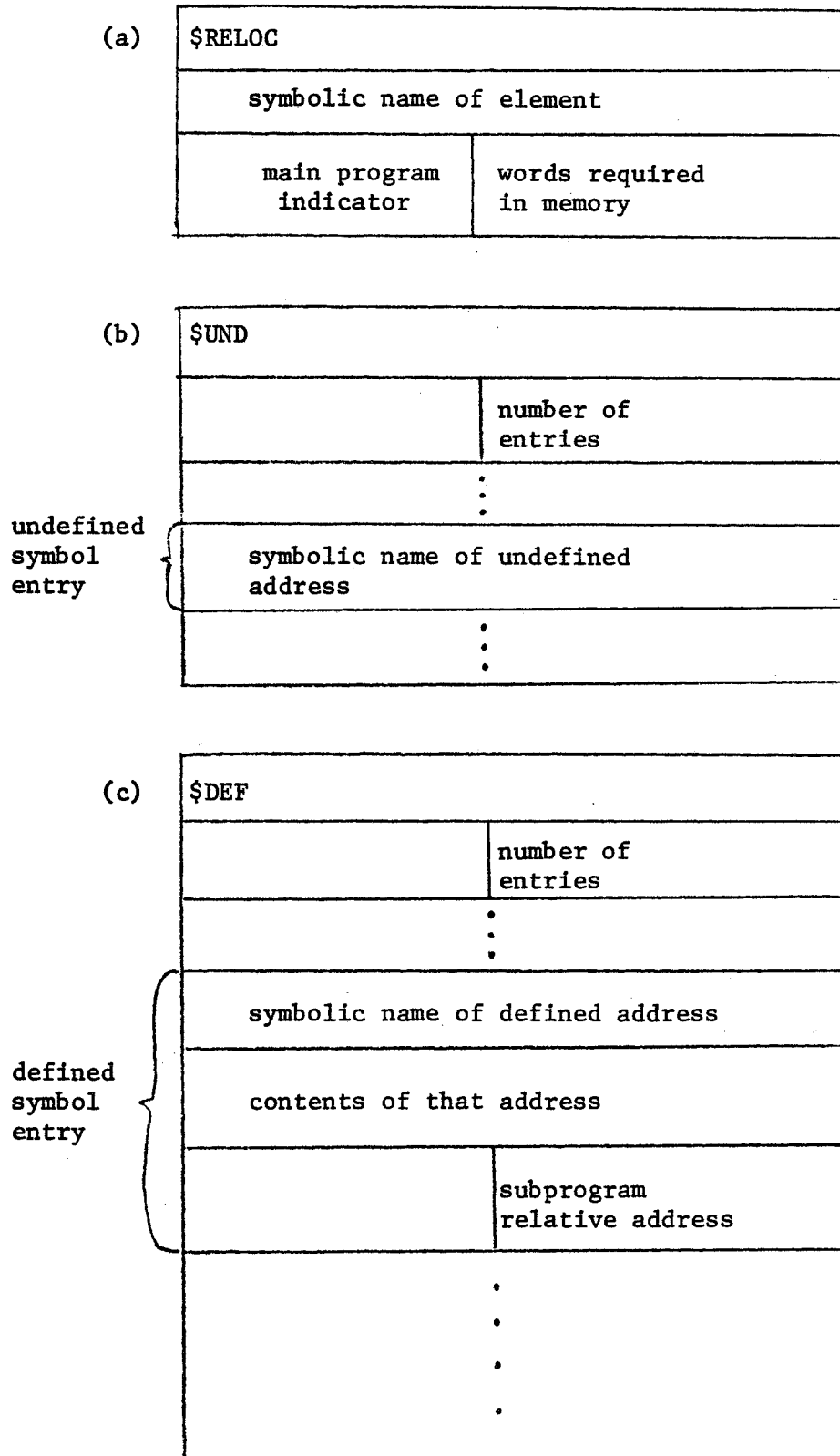


Figure 3 The Relocatable Element

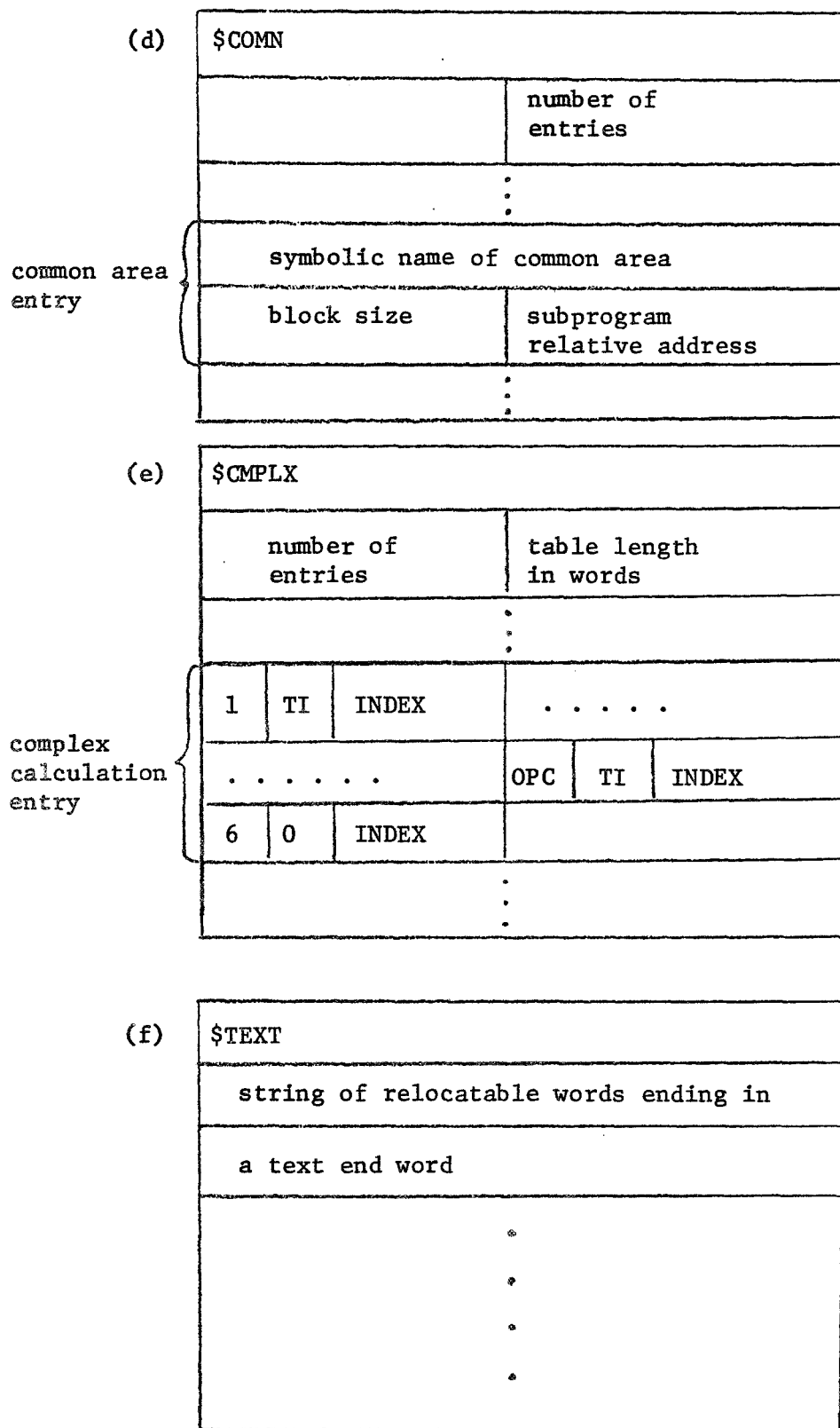


Figure 3 (continued)

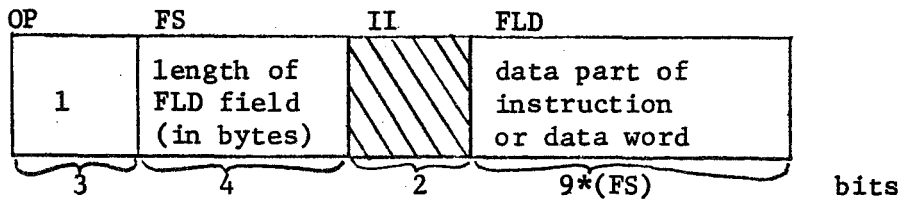
relocatable code is the intermediate form of the machine language instructions and data that results from the translation of an assembly language subprogram or of a procedural language subprogram. The relocatable code is contained in \$TEXT control records. The symbolic address tables are identified by four control words, \$UND, \$DEF, \$CCMN, and \$CMFIX.

The \$TEXT control word identifies a control record containing relocatable code (Figure 3(f)). A machine language instruction is usually made up of a non-address part (e.g., the op-code part) and an address part. Since relocatable translation requires only the adjustment of addresses, it is only necessary to distinguish between the address part and the non-address part of the instructions. Therefore, a relocatable code consists of a sequence of relocatable words. Each relocatable word contains an address part or a non-address part of an instruction. In this context, a data word can be viewed as an instruction without an address part. Figure 4 shows the six formats (called A, B, C, D, E, and F) of the relocatable words. Each format has up to five fields: OP, FS, II, FLD, and INC. The FLD field contains data, or the address part of an instruction, or an index to a table. The FS field contains the length of the FLD field in bytes (for convenience, a byte is defined here as 9 bits). The OP field identifies the FLD field as:

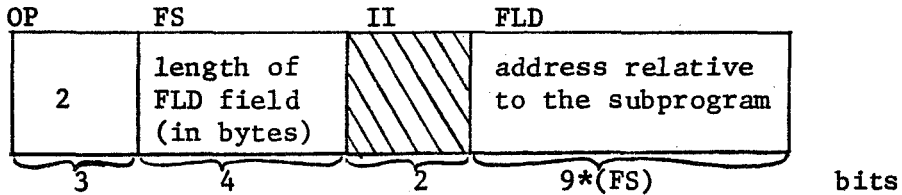
- (a) Data (CP=1).
- (b) Relative address (OP=2). This address references a location within the subprogram relative to the subprogram address. The FLD field contains a relative address.
- (c) Common address (CP=3). This address references a data area which is common to several subprograms. The FLD field contains an index to the common area list.

Figure 4 Relocatable Word Formats

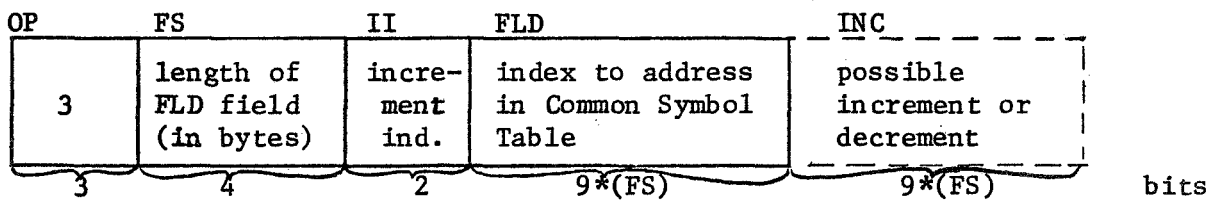
31



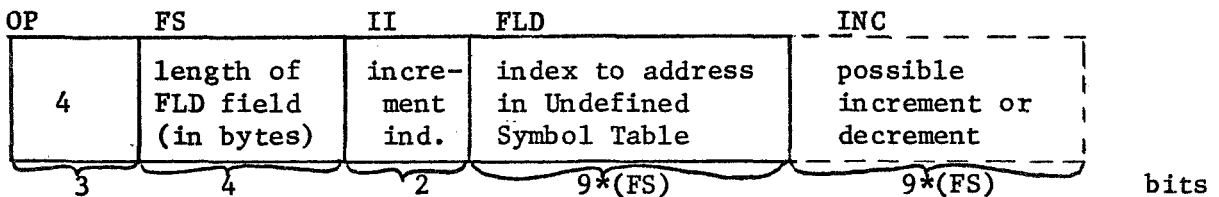
(a) Format A, indicating data



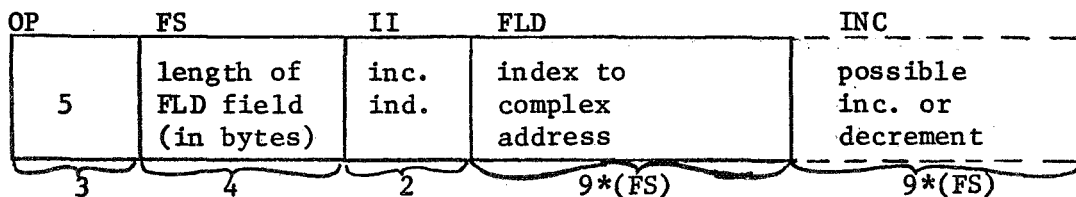
(b) Format B, indicating relative address



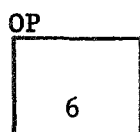
(c) Format C, indicating common data address



(d) Format D, indicating external address



(e) Format E, indicating complex address



(f) Format F indicating the end of relocatable code

- (d) External\_address (OP=4). This address references an entry point in another subprogram. The FLD field contains an index to the undefined symbol list.
- (e) Complex\_address (OP=5). This address is an arithmetic combination of external, common, or other complex addresses. The FLD field contains an index to the complex address list.
- (f) End\_of\_relocatable\_code (OP=6).

The above complex address, common address and external address words reference locations outside of the subprogram. These addresses are symbolic and are stored in the symbolic address tables to be described subsequently. The II and INC fields are provided to give a numerical increment to these symbolic addresses. If II is equal to 1 or 2, the address is incremented or decremented, respectively, by the contents of INC. If II is equal to 0, there is no INC field.

Figure 5(a) shows an example of a relocatable code where the fields are separated by vertical lines and the numbers of octal. Words 1,2,3,5,7,9,11,13, and 15 are relocatable words with data. Words 6,8,12,14, and 16 are those with relative addresses (OP=2). Word 17 is one with a common address (OP=3). Words 4,10, and 18 are those with external addresses (OP=4). Word 19 is the one indicating the end of the relocatable code (OP=6). There are no complex addresses in this example. Note that the relocatable words in Figure 5(a) are of different lengths. Though they are shown as left-justified, they are actually a string of bytes as shown in Figure 5(b). It is in the format of Figure 5(b) that the relocatable words are stored in the memory.

Symbolic address tables of a relocatable element contain all the symbolic addresses that are required to link subprograms together and those address calculations that

(a) in relocatable words (octal)

A	PZE	1.0	1 2 0	2 0 0 4 0 0 0 0 0 0 0 0	1
B	PZE	1.0	1 2 0	2 0 0 4 0 0 0 0 0 0 0 0	2
MAIN	TSX	COS,4	1 1 0	0 0 7 4 0 0	3
			4 1 0	4 0 0 0 0 1	4
	PZE	A	1 1 0	0 0 0 0 0 0	5
			2 1 0	0 0 0 0 0 0	6
	STO	A	1 1 0	0 6 0 1 0 0	7
			2 1 0	0 0 0 0 0 0	8
NEXT	TSX	SIN,4	1 1 0	0 0 7 4 0 0	9
			4 1 0	4 0 0 0 0 0	10
	PZE	B	1 1 0	0 0 0 0 0 0	11
			2 1 0	0 0 0 0 0 1	12
	FAD	A	1 1 0	0 3 0 0 0 0	13
			2 1 0	0 0 0 0 0 0	14
	AXT	NEXT,4	1 1 0	0 7 7 4 0 0	15
			2 1 0	4 0 0 0 0 5	16
	TIX	SUBR,3,BLOCK+10	3 1 1	2 0 0 0 0 0	17
				0 0 0 0 0 5	
			4 1 0	3 0 0 0 0 2	18
			6 0 0		19

OP
FS
IT
FLD
INC

(b) as a series of memory words (octal)

1 2 0 2 0 0 4 0 0 0 0 0
0 0 0 1 2 0 2 0 0 4 0 0
0 0 0 0 0 0 1 1 0 0 0 7
4 0 0 4 1 0 4 0 0 0 0 1
1 1 0 0 0 0 0 0 0 2 1 0
0 0 0 0 0 0 1 1 0 0 6 0
1 0 0 2 1 0 0 0 0 0 0 0
1 1 0 0 0 7 4 0 0 4 1 0
4 0 0 0 0 0 1 1 0 0 0 0
0 0 0 2 1 0 0 0 0 0 0 0
1 1 0 0 3 0 0 0 0 2 1 0
0 0 0 0 0 0 1 1 0 0 7 7
4 0 0 2 1 0 4 0 0 0 0 5
3 1 1 2 0 0 0 0 0 0 0 0
0 0 5 4 1 0 3 0 0 0 0 2
6 0 0 0 0 0 0 0 0 0 0 0

Figure 5 Example of a relocatable text

depend upon addresses that are unknown to the assembler or compiler. There are four symbolic address tables:

(a) The\_undefined\_symbol\_list (Figure 3(b)). This is one or more control records containing the control word \$UND in the first word, the number of entries in the second word, and followed by a list of symbolic addresses that were not resolved by the assembler or compiler and therefore assumed to be external to the subprogram. There is one symbolic address to the word.

(b) The\_defined\_symbol\_list (Figure 3(c)). This is one or more control records containing the control word \$DEF in the first word, the number of entries in the second word, and followed by a list of symbolic addresses from the subprogram that are typed global by the assembler or compiler. These are the addresses that will correspond with undefined symbol entries in other relocatable elements. Each entry consists of three words, the symbolic name of the address, the contents of the word at that address if it is data (zero otherwise), and the subprogram relative address corresponding to the symbolic address.

(c) The\_common\_area\_list (Figure 3(d)). This is one or more control records containing the control word \$COMN in the first word, the number of entries in the second word, and then a list of entries representing common areas specified in the subprogram. Each entry in the list consists of two words, the first of which contains the symbolic name of the common area. If the subprogram defines the contents of the common area, the second word contains a zero first half and the subprogram relative address of the common area in the second half. If the subprogram does not contain the common area, the second half of the second word is zero, and the first half contains the number of words specified for the common area by the subprogram.

(d) The\_complex\_address\_list (Figure 3(e)). This is one or more control records specifying a number of arithmetic operations to be made between global addresses, common area addresses, previous complex addresses, or the contents of global addresses. The first word contains the control word \$CMPLX; the first half of the second word contains the number of complex addresses calculated in the table; and the second half of the second word contains the number of words in the table of complex calculations that follow. The following list of words contains a string of half-word instructions for each address to be computed. If there is more than one \$CMPLX control record, only the first record specifies the number of complex addresses (the first half of word two).

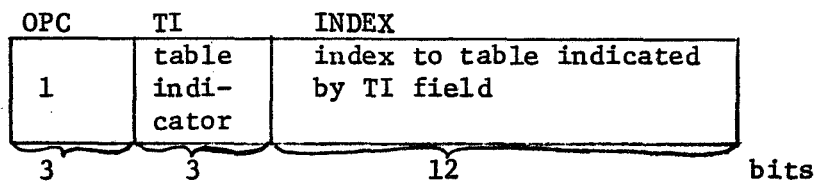
Figure 6 shows the format of the complex calculation word. The field CPC indicates one of six operations: load, add, subtract, multiply, divide, and store (all arithmetic is integer). The field TI indicates one of five sources of operands: the undefined symbol list to obtain a global address (TI=1) or the contents at a global address (TI=2), the common area list to obtain a common area address (TI=3), the complex address list to obtain the result of a previous complex calculation (TI=4), or the value of the field INDEX (TI=5). The field INDEX is used to index the specified table or as the operand.

Examples of relocatable elements appear in Figures 7, 8, and 9. Figure 7 shows a relocatable element that has four control records. The \$RELOC control record shows that the subprogram has symbolic name MAIN, that it is the main subprogram, and that it requires 10 words of memory in executable form. The \$UND control record shows that there are three undefined symbols SIN, COS, and SUBR. The \$CCMN control record shows that there is one common area BLOCK referenced in the subprogram and that it requires 8 words.

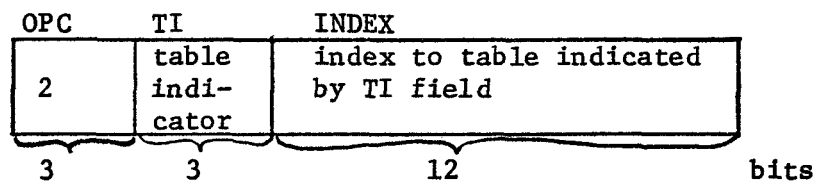


Complex calculation words

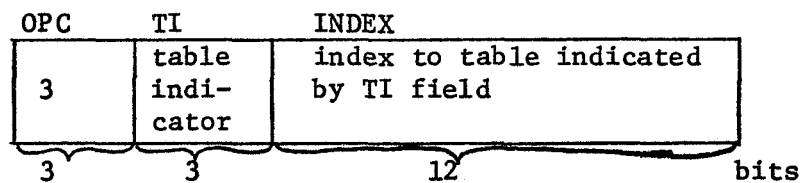
(a) load instruction (begin calculation)



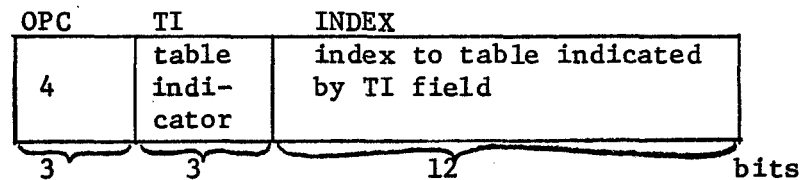
(b) add instruction



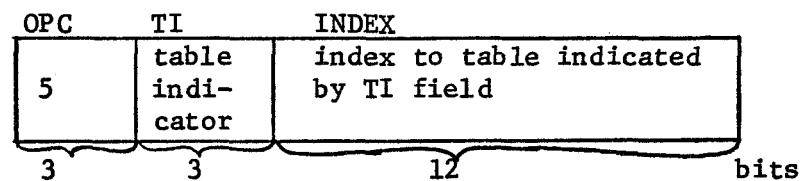
(c) subtract instruction



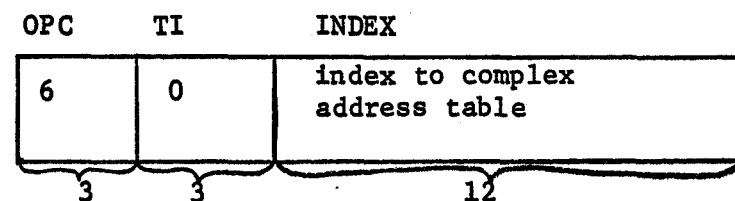
(d) multiply instruction



(e) divide instruction



(f) store instruction (end calculation)



## Table indicator values

1	undefined symbol list- address
2	undefined symbol list- value
3	common symbol list
4	complex address list
5	value of INDEX field

Figure 6

Figure 7 Example of user input

\$	R	E	L	O	C	
S	U	B	R	T	N	
		0,		2	0	0
\$	U	N	D	b	b	
						3
S	I	N	b	b	b	
C	Ø	S	b	b	b	
C	N	S	T	N	T	
\$	D	E	F	b	b	
						2
C	N	S	T	N	T	
0	0	0	0	0	0	7
						7
						7
						7
						0
S	U	B	R	b	b	
						0
						1
						0
\$	C	O	M	N	b	
						1
B	L	Ø	C	K	b	
		1	2,			0
\$	C	M	P	L	X	
		2,				3
1,2,		2,3,1,				0
6,0,		0,1,2,				2
3,1,		1,6,0,				1
\$	T	E	X	T	b	
		.				
		.				
		.				

Figure 8 Example of library subroutines

\$	R	E	L	O	C
S	I	N	C	Ø	S
			0,		5 0
\$	D	E	F	b	b
					2
C	Ø	S	b	b	b
					0
					0
S	I	N	b	b	b
					0
					5
\$	T	E	X	T	b
		.			
		.			
		.			
		.			

Figure 9 Example of Library Subroutine (SINCOS)

The \$TEXT control record contains a 16 word string of relocatable code. Figure 8 shows a relocatable element composed of six control records. The subprogram name is SUBRTN. There are three undefined addresses referenced within it, SIN, COS and CNSTNT. There are two global addresses within it, CNSTNT and SUBR, and one common area, BICCK, is referenced. In addition, there is a complex calculation control record \$CMFEX. This indicates that the computations for two complex addresses are contained in the following three words of half-word instructions. The first half-word instruction has CFC=1, TI=2, and INDEX=2. This translates (by the format in Figure 6) to 'load the value of the symbolic address that is the third entry in the undefined symbol list (the index begins at zero)' or 'load the contents of the word at symbolic address CNSTNT.'

There are two sources of input to the loader, the user and the library. The user input to the loader is composed of the \$ABSIT control record, followed by a set of relocatable elements. Figure 7 is a one element example of this. If memory overlay is desired, the set of relocatable elements may be broken into subsets by \$SEG control records. In this case, the overlay structure is specified by the list of predecessor segments, as mentioned before. Figure 10 shows a list of \$SEG control records that define the overlay structure diagrammed in Figure 1(c). It is important to realize that, although each segment lists only the segments that immediately precede it in the structure, the overlay structure is completely defined. For example, segment J lists segments G, I, and D as immediate predecessors. However, each of these in turn lists its predecessors, and so on, up to segment MAIN which is the main (or root) segment (i.e., it has no predecessors). It should also be recognized that the order in which program segments are input is not specified. The loader performs its own ordering internally.

```

$SEG MAIN
{relocatable elements for segment MAIN}

$SEG C (MAIN)
{relocatable elements for segment C}

$SEG D (MAIN)
{relocatable elements for segment D}

$SEG G (B)
{relocatable elements for segment G}

$SEG I (C)
{relocatable elements for segment I}

$SEG F (A)
{relocatable elements for segment F}

$SEG E (A)
{relocatable elements for segment E}

$SEG H (B,C)
{relocatable elements for segment H}

$SEG A (MAIN)
{relocatable elements for segment A}

$SEG J (G,I,D)
{relocatable elements for segment J}

$SEG B (MAIN)
{relocatable elements for segment B}

```

Figure 10

The other source of input is the subprogram library. If, after all of the user subprograms are input from I/O unit INU, some external addresses are as yet undefined, an attempt is made to resolve these in the subprogram library. This library is accessed by searching an index of symbolic global addresses that have been culled from the subprogram library. This index exists as a chain of one or more records on I/O unit LIB1.

Figure 11 shows the format of the library index table. Each record represents a table. The symbolic global addresses are ordered lexicographically by name. Each table is headed by four words that give the lower and upper bounds of global addresses contained in the table, the mass storage location of the next library index (if any), and the number of entries in the table. The entries are composed of two words: the first is the symbolic global address; and the second is the mass storage location where the relocatable element which contains it can be found. Figure 12 shows an example of a library index table. The lower and upper bounds (0 and 7777777777) show that this index contains all symbolic addresses in the library. These include global addresses CONST, CCS, SIN, and SUBR. These addresses reside in the relocatable elements at mass storage locations LIBLC1, LIBLC2, LIBLC2, and LIBLC1, respectively.

### 3.1.3 Comparison With Existing Loader Input

The chief differences between the input specification given above and that of existing loaders are in the format of the relocatable code, the structure of the undefined symbol table, and the library index. For instance, the definition of the relocatable word given above is unique.

lower bound of entry point names in the table	
upper bound of entry point names in the table	
mass storage location of the next library table	
number of entries in this table	
.	
.	
.	
symbolic entry point name	} library table entry
mass storage location of subprogram in which the entry point occurs	
.	
.	
.	

Figure 11 Library Index



					0
7	7	7	7	7	7
					0
					4
C	N	S	T	N	T
L	I	B	L	C	1
C	Ø	S	b	b	b
L	I	B	L	C	2
S	I	N	b	b	b
L	I	B	L	C	2
S	U	B	R	b	b
L	I	B	L	C	1

Figure 12      Example of a Library Index

The traditional method of relocatable coding is to provide a table of descriptors, each describing the relocation of one or more words of the program text. For example, the IBM 7090/7094 loader packs seven five-bit descriptors into one 36-bit word to describe the relocation of seven words. Each five-bit descriptor specifies the relocation required for the two address fields in the IBM instruction word. The UNIVAC 1108 relocatable output routine (ROB) produces a variable length string of binary information that is used to relocate one or more instruction words that follow it.

The second point of comparison is the undefined symbol table. A number of operating systems employ compilers and assemblers that perform all inter-subprogram jumps by branching to a single location within the subprogram (called the transfer vector) that contains the address of the desired subprogram. Therefore, their loaders contain no inter-subprogram transfers from within the text; instead, they translate the undefined symbol table into a transfer vector table. As the loader described herein could make no such assumptions about the compiler, this format could not be adopted (note, however, that this does not preclude the possibility of such a table).

Another loader, that of the CDC 6600, structures the links to the undefined symbol table in reverse [9]. Instead of each external reference pointing to the undefined symbol table, the table contains a pointer to the first address field that references it, that address field contains a pointer to the second, and so on, with the last address field containing an end-of-chain marker. In this manner, inter-subprogram addresses are easily located.

The third point of comparison is the library index. It is used chiefly because it is a simple implementation, while a number of loaders provide multiple-library search procedures with complex command languages. To include such

procedures would require detailed description of file systems, which is outside the province of this paper.

### 3.2 The Output

The absolute element is the form that the user program takes in order to exist outside of memory. It is structured to be easily loaded into memory for execution. It consists of a resident part that is loaded initially and remains in memory throughout execution, and, if necessary, a series of non-resident parts that are loaded during execution as required. The algorithm assumes that there is a standard system I/O unit for temporary storage of the non-resident segments.

The resident portion consists of the main segment, all common areas, the segment loading routine (\$LINK\$), the linkage table, and the segment table. The non-resident portion consists of all overlay segments.

In order to facilitate loading of the program, the absolute program is blocked into logical blocks. A logical block represents one overlay segment or the resident portion. It is assumed that the absolute element is to be stored lineally on mass storage as a series of records. Each logical block is composed of a series of records. Each record represents a contiguous piece of memory. However, the records need not be ordered by memory location.

The problem of orienting in memory a set of records representing a non-contiguous set of memory blocks is met by the method of scatter loading. At the beginning of each record of contiguous memory locations is the address of the first location. Loading becomes a simple process of

interpreting the address and transmitting the following block of words to that location and locations following.

The structure of the absolute element is shown in Figure 13. Heavy lines separate the logical blocks. The header block identifies the absolute element, the overlay segments follow it, and the resident portion of the program completes the absolute element.

The header block differs from all the other records in that it is not a part of the program. It provides information to the memory load routine. Figure 14(a) shows the format of the header block. The first word contains the symbolic name of the absolute element. The second word contains the overlay indicator and the main segment flag. If the overlay indicator is non-zero, overlay segments will follow the header block. The third word gives the address of and the number of words in the undefined common areas. The last word gives the absolute address of the program to which control is to be transferred once the resident part of the program is loaded (i.e., the starting address).

Each record of executable code appears as in Figure 14(b). The first word contains the segment indicator in the first half and the address at which the block of executable code is to be loaded in the second half. The segment indicator is a flag that is zero for non-resident portions of the program and non-zero for the resident portion of the program.

The overlay linkage table and the segment table are shown in Figures 14(c) and 14(d). The structure of the first word is the same as above except that both have non-zero flag fields because they are always part of the resident portion of the program. The overlay segment table consists of a two-word entry for each global address in a non-resident segment that is referenced from another

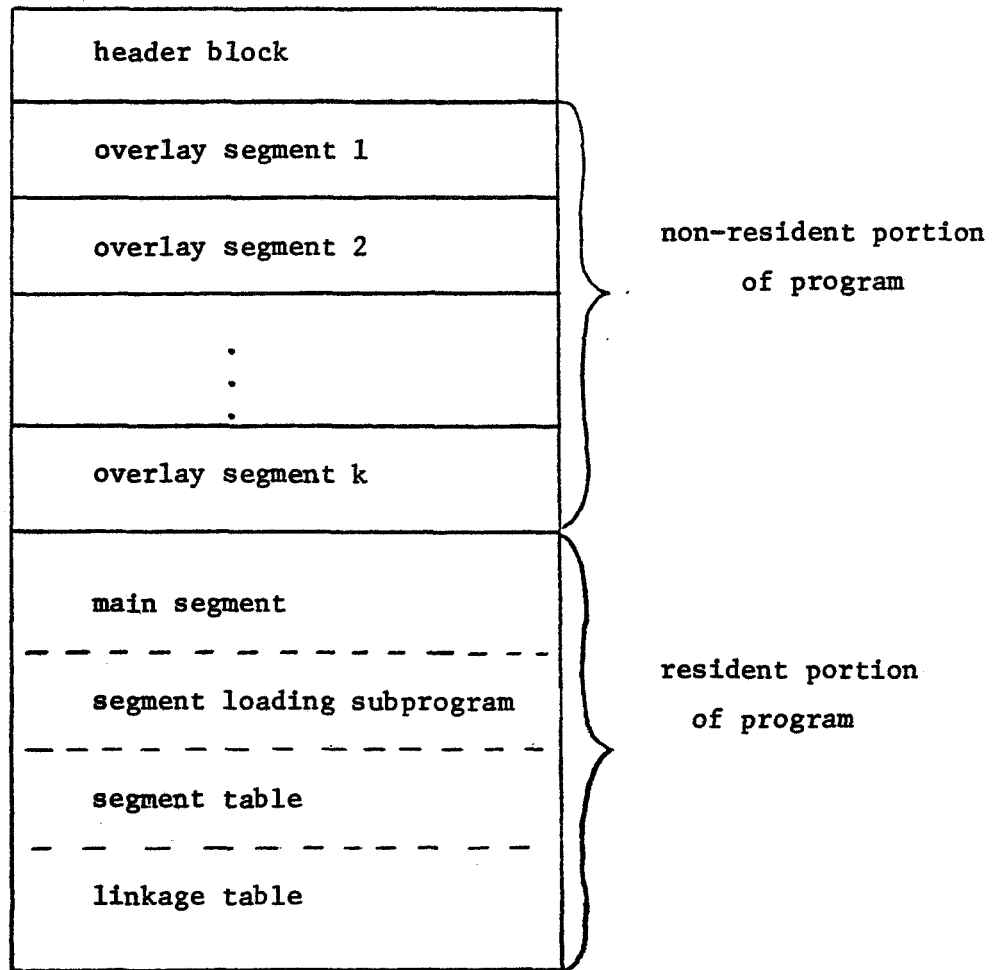


Figure 13 The Structure of the Absolute Element

- (a) **Absolute element header block**
- The segment table contains a one-word entry for each non-resident segment, indicating the location of the segment in the absolute element header block and number of records

symbolic name of the absolute element	
overlay indicator	main segment indicator
absolute address of common area	number of locations in common area
starting address	

- (b) **Executable code block**
- With a zero flag field, are transferred are transferred by the system. Once the main segment flag is set, the records are scattered into core.

segment indicator	memory address at which to load this record
An example of the absolute element is shown in Figure 15. The absolute element SAMLE has a 10 word common area beginning at 1000. The starting address is 1010. There are no overlay segments, but the main segment flag (FFFF) is shown anyway. There are three records of 12, 200, and 20 words. The records are to be loaded at 1000, 1010, and 1030 respectively.	

- (c) **Overlay linkage table**
- The records are to be loaded at 1000, 1010, and 1030 respectively.

segment indicator	memory address at which to load the table
:	
branch instruction to \$link\$	
segment number	desired absolute address
:	
:	

- (d) **Segment table**

segment indicator	memory address at which to load the table
location and size of first overlay segment	
:	
location and size of last overlay segment	

Figure 14 Detailed Block Diagrams of the Absolute Element Records

segment. The segment table contains a one-word entry for each non-resident segment, indicating the location of the segment in the absolute element and number of records required for the segment.

The loading of the absolute element is relatively simple. The header record identifies the location of the undefined common areas and the number of locations to be set to zero. Then the following overlay segment records, those with a zero flag field, are transferred to the temporary storage specified by the system. Once the main segment flag is encountered, the records are scatter loaded into core.

An example of the absolute element is shown in Figure 15. The absolute element SAMPLE has a  $10_8$  word common area beginning at  $1000_8$ . The starting address is  $1014_8$ . There are no overlay segments, but the main segment flag ( $555555_8$ ) is shown anyway. There are three records of  $12_8$ ,  $200_8$ , and  $50_8$  words. The records are to be loaded at locations  $1012_8$ ,  $1024_8$ , and  $1224_8$ , respectively.

12 words

S	A	M	P	L	E
0	0	0	0	0	5
5	5	5	5	5	5
1	0	0	0		1
2					
1	0	1	4		0

5	5	5	5	5	5	0	0	1	0	1	2
2	0	0	4	0	0	0	0	0	0	0	0
2	0	0	4	0	0	0	0	0	0	0	0
0	0	7	4	0	0	4	0	1	2	2	4
0	0	0	0	0	0	0	0	1	0	1	2
0	6	0	1	0	0	0	0	1	0	1	2
0	0	7	4	0	0	4	0	1	2	3	1
0	0	0	0	0	0	0	0	1	0	1	3
0	3	0	0	0	0	0	0	1	0	1	2
0	7	7	4	0	0	4	0	1	0	1	7
2	0	1	0	0	5	3	0	1	0	3	4

200 words

5	5	5	5	5	5	0	0	1	0	2	4

50 words

5	5	5	5	5	5	0	0	1	2	2	4

Figure 15 Example of an  
Absolute Element



#### 4.0 The Algorithm

The algorithm for the loader consists of three phases. In the first phase, all user relocatable elements are input along with any segment control records. The data structures representing the input are constructed, including lists of referenced global addresses and a table of defined global addresses. The subprogram library is searched to find any global addresses that are referenced but are not defined in the user input. The library subprograms necessary to complete the program are input and added to the tables. Throughout the first phase, the relocatable code is stored on mass storage.

In the second phase, the common areas and the subprograms of each segment are allocated absolute memory addresses (termed the subprogram\_\_\_address). Once a subprogram has been allocated its subprogram address, the global addresses within it are assigned absolute addresses. When the memory allocation is complete, the complex addresses are calculated.

In the third phase the relocatable code is input from mass storage a record at a time and translated to executable code. The subprogram relative addresses are added to the subprogram address, each external reference to a global address is replaced with the absolute address assigned to the global address, the references to common areas are replaced with the absolute addresses assigned to the common areas, and each reference to a complex address is replaced with the value of that complex address. All address references between segments are made through a linkage table constructed in this phase. The executable code is formed as scatter-load records and placed on the output unit.

The algorithm is described in detail in the subsequent sections. These sections describe the data structures necessary to represent the information in memory, the characteristics of the necessary input/output routines, and the three phases of the algorithm. The design of the algorithm, including input, output, and data structures, is based upon existing loaders [4] [5] [7] and the description of loaders found in the literature [10] [11] [12] [13].

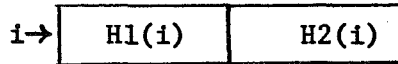
#### 4.1 Structure Of The Data In Memory

This section presents the data structures that are used within the loader to store the data and to provide the easiest access to it for all the functions of the loader. Once the structures have been detailed, some alternatives will be considered briefly. It is assumed that the reader is familiar with data structures such as a queue, a sequential list, and a linked list (e.g., as described in Knuth [14]). However, a few basic definitions are in order. The information in a data structure consists of a set of nodes. A node will also be called an entry, cell, or a descriptor. Each node consists of one or more consecutive words of memory. The address of a node is the memory location of its first word and is also referred to as a pointer, link, or reference. A node has one or more named parts called fields.

For example, a single word is a node. Within this paper the fields of a single word will be named as shown in Figure 16. The full word at address  $i$  is named  $W(i)$ , the two halves are named  $H1(i)$  and  $H2(i)$ , and the quarter words are named  $Q1(i)$ ,  $Q2(i)$ ,  $Q3(i)$ , and  $Q4(i)$ . Within the context of this paper each word is assumed to contain two

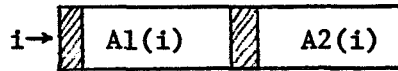


$W(i) \equiv$  contents of the memory word at address  $i$



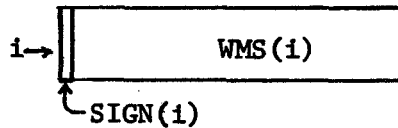
$H1(i) \equiv$  contents of the first half of the word at  $i$

$H2(i) \equiv$  contents of the second half of the word at  $i$



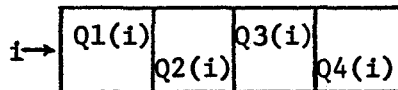
$A1(i) \equiv$  contents of the address portion of the first half of the word at  $i$

$A2(i) \equiv$  contents of the address portion of the second half of the word at  $i$



$WMS(i) \equiv$  contents of the word at  $i$  except the sign bit

$SIGN(i) \equiv$  contents of the sign bit of the word at  $i$



$Q1(i) \equiv$  contents of the first quarter of the word at  $i$

$Q2(i) \equiv$  contents of the second quarter of the word at  $i$

$Q3(i) \equiv$  contents of the third quarter of the word at  $i$

$Q4(i) \equiv$  contents of the fourth quarter of the word at  $i$

Figure 16 Field Descriptions of a Memory Word

address fields,  $A1(i)$  and  $A2(i)$ , located in the right most part of each half-word. The left-most bit of each word is named  $SIGN(i)$ , and the remaining portion of the word is named  $WMS(i)$ .

The loader has the main memory available as a storage area. In addition, the system provides temporary storage through the I/O processor, 'secretary' (SEC), which is described in detail in a subsequent section. The main memory is random access word-addressable, while the secretary can store a sequential series of variable length records at any one of a large number of symbolic mass storage locations (i.e., the secretary provides the effect of a random access mass storage file system whether one exists or not).

The memory requirements of the loader are of four types: (a) a fixed number of variables and pointers in three tables, (b) a set of expanding tables, (c) an input buffer area in which to place input records, and (d) a multi-use area that can be used to store temporary information during the first part of the algorithm and as an output buffer in the latter part of the algorithm. Figure 17 shows the basic form of the required memory area. The three tables in the first section contain the pointers and variables, the table descriptors for all the following tables, and the hash table. The pointers and variables will be introduced throughout the algorithm and their specific location in the table is inconsequential. The hash table will be explained in detail with the defined symbol table. The table and buffer descriptors are introduced in Figure 18. Each descriptor consists of four fields in two words C, F, L, and N. Each descriptor is defined by its address. In order to distinguish the descriptors, abbreviations for the tables will be used to name the descriptor addresses.

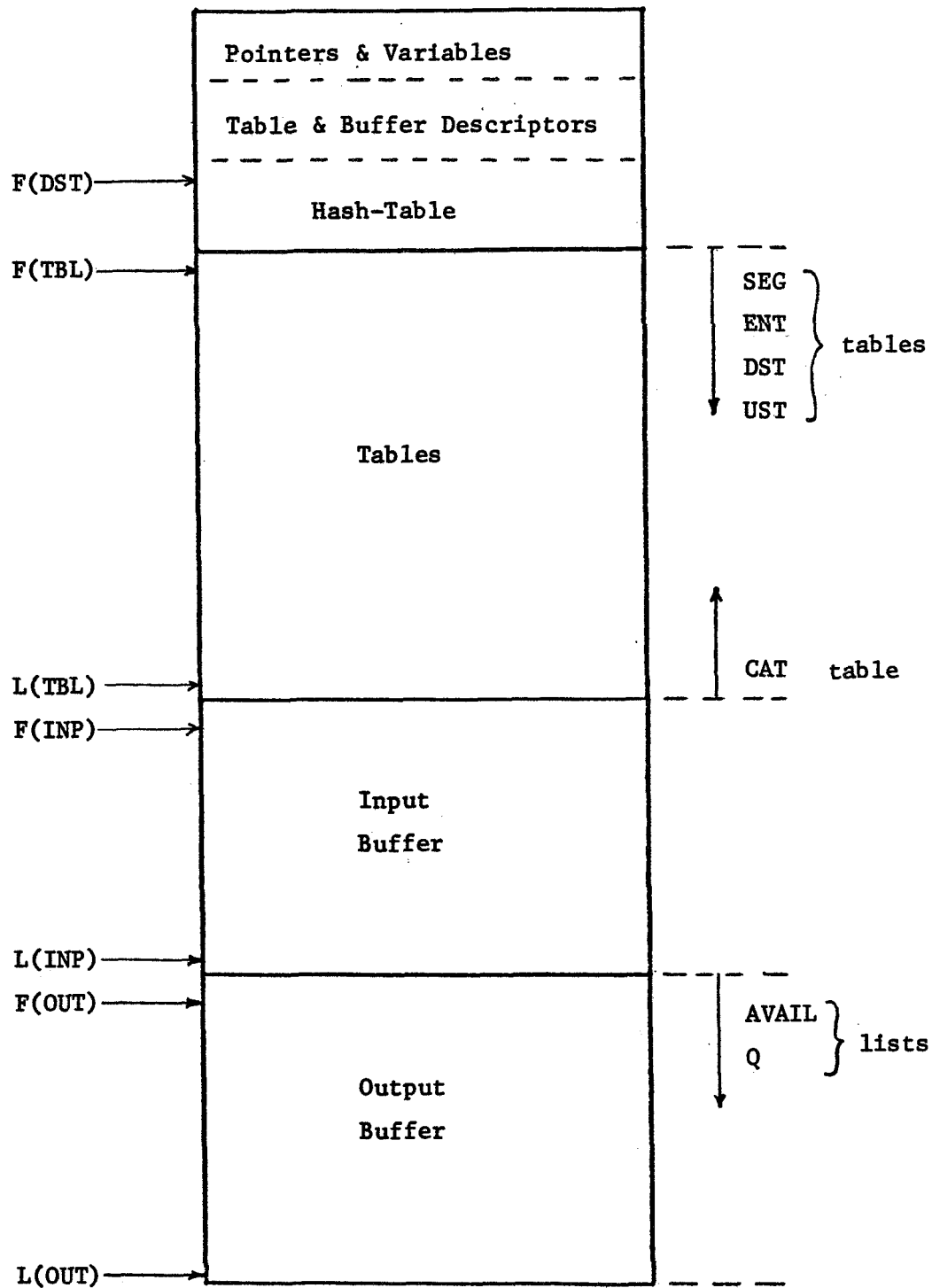


Figure 17 Orientation of Memory Tables

## Tables and Buffers

TBL table area for:

SEG segment descriptor table

ENT element descriptor table

DST defined symbol descriptor table

UST table of unresolved symbols

ELT overlay linkage table

CAT common area table

INP input buffer

OUT output buffer and temporary table area for:

AVAIL table of available cells

Q library queue

## Table Descriptors

For each of the above tables and buffers there is a two word descriptor of the following format:

C	F
L	N

C≡ address of the current entry in the table or buffer

F≡ address of the first entry in the table or buffer

L≡ address of the last entry in the table or buffer

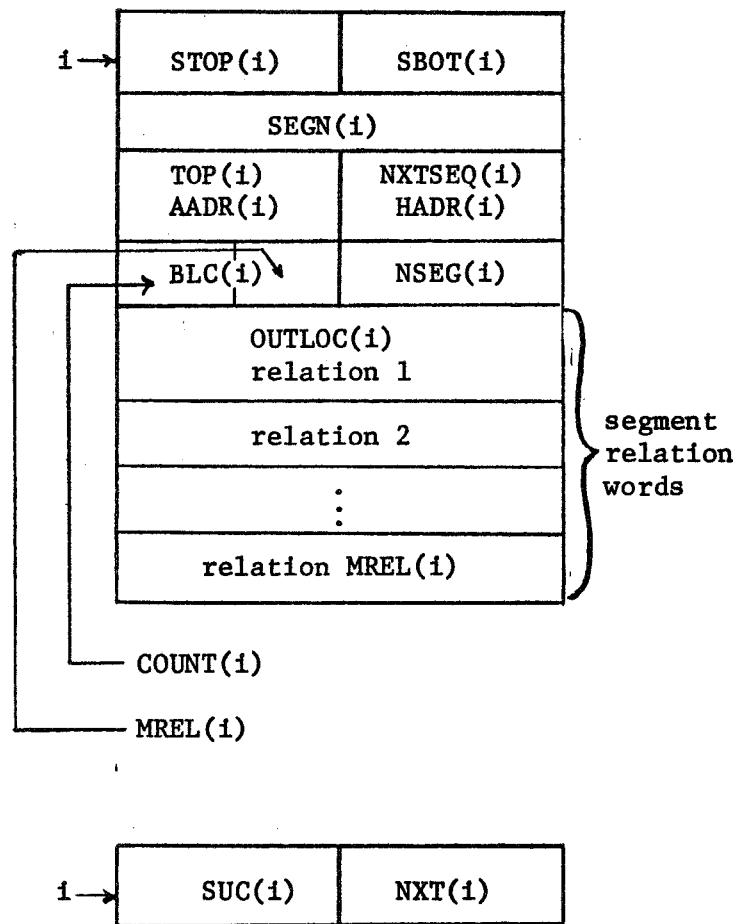
N≡ number of entries in the table or buffer

Figure 18

The expanding tables are placed in the area of memory defined by the F and L fields of the TBL descriptor. Within this area are the segment descriptor table (SEG), the element descriptor table (ENT), the defined symbol table (DST), the table of unresolved symbols (UST), the overlay linkage table (EIT), and the common area table (CAT). Each of these tables has a descriptor. The limits of the input buffer and the output buffer are defined by the F and L fields of the INP descriptor and OUT descriptor, respectively. Within the output buffer area are two tables, a table of available two-word cells (AVAIL) and a queue of required library subprograms (Q).

The philosophy of the algorithm presented is to form a data structure in memory that represents the memory allocation of the program. Once this structure is complete, then the translation of relocatable code to executable code can be performed. This requires that all user input and library input be represented in memory. However, the relocatable code ( \$TEXT control records) is placed on temporary storage and represented by storage location alone.

Each segment is represented by a segment descriptor. There is at least one segment descriptor, whether or not \$SEG control records are input. The fields are defined specifically in Figure 19. However, several fields serve dual functions and are therefore assigned two names. For example, for a segment descriptor located at address i, H1(i+1) is named TCP (i) and ADDR (i), and H1(i+4) is named EIC (i) while Q1(i+4) and Q2(i+4) represent the same field and are named CCUNT (i) and MREL (i). Segment descriptors are linked together by the NXTSEQ field in the order they were input and by the NSEG field in the order that they are to be output. In addition, each segment points to the beginning and end of a chain of element descriptors (representing the subprograms that belong to it) through the



STOP(i)	address of the descriptor for the first element in the segment
SBOT(i)	address of the descriptor for the last element in the segment
SEGN(i)	symbolic name of the segment
TOP(i)	address field used in the ordering of the segment descriptors
AAADR(i)	absolute address assigned to the first element in the segment
HADR(i)	absolute address of the first word following the segment
NXTSEQ(i)	address of the descriptor for the segment following the segment, order input (NXTSEQ)
NSEG(i)	order processed (NSEG)
COUNT(i)	number of direct predecessor segments for the segment
MREL(i)	number of segment relation words
BLC(i)	number or records required to store the executable code for the segment
OUTLOC(i)	location of the first record of executable code for the segment
SUC(i)	address of the segment that represents the next in the chain of successor segments for the segment
NXT(i)	address of the next relation word

Figure 19 Segment Descriptor

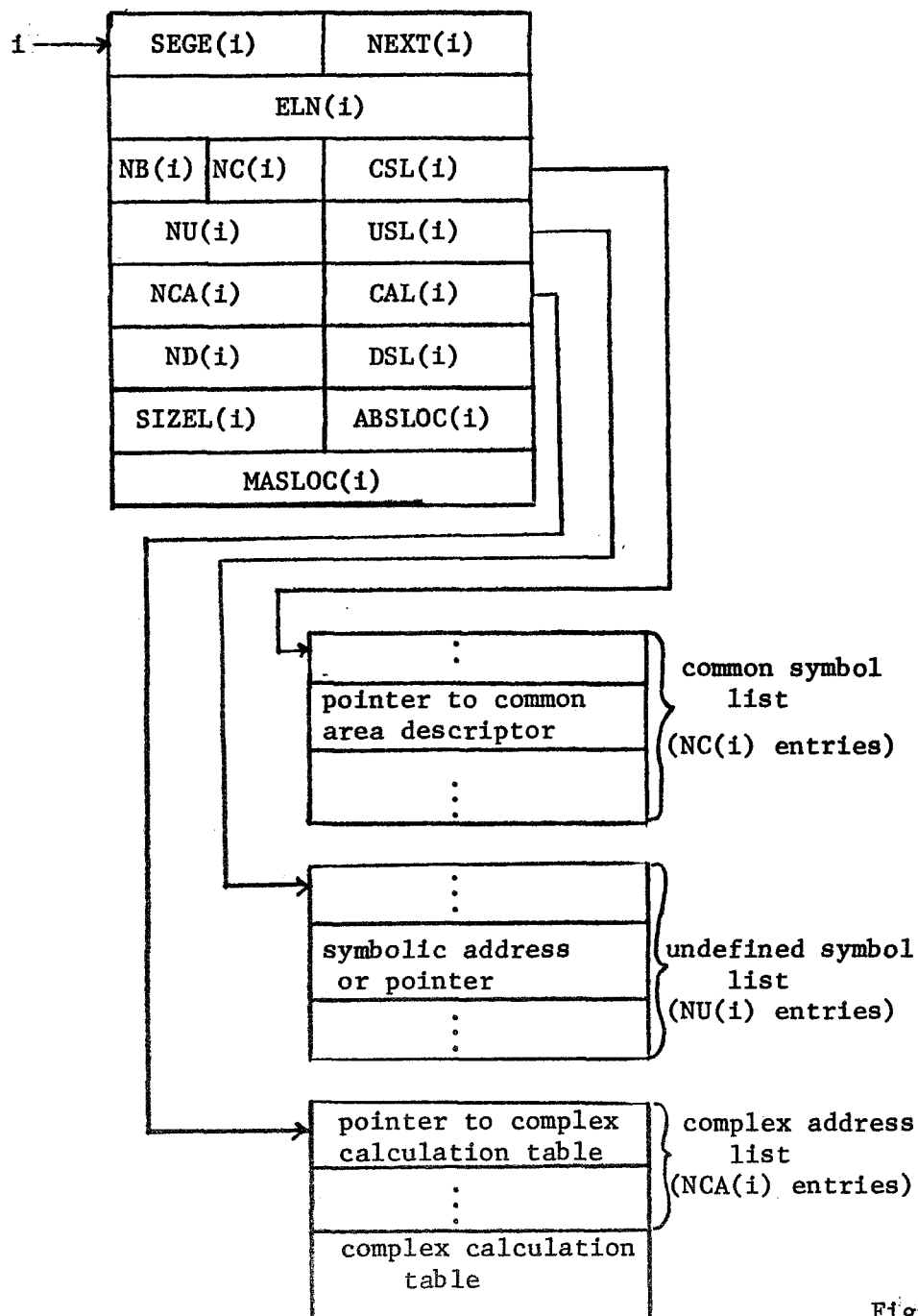


STCF and SECT fields. The list of segment relation words at the end of the segment descriptor are used to represent the predecessors listed on the \$SEG control record.

Each subprogram is represented by an element descriptor. Each element descriptor consists of an eight word block followed by three lists: the common symbol list, the undefined symbol list, and the complex address list. Figure 20 describes the descriptor and shows how the main block points to the three lists through the CSI, USI, and CAI fields. Each element descriptor is linked to the next element descriptor in the segment chain by the next field. The SEGE field of the element descriptor points to the segment descriptor to which it belongs, and the DSL field points to a chain of global symbols that occur within the subprogram.

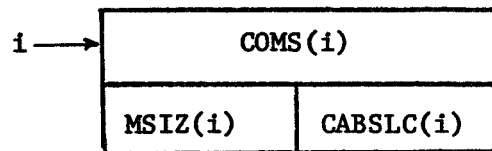
The common area descriptor is shown in Figure 21. There is one two-word descriptor for each common area. The common area descriptors are stored sequentially from I(TBI) toward F(TBI).

The defined symbol descriptor is shown in Figure 22. There is one five-word descriptor for each global address specified in a relocatable address. Each descriptor points to the element descriptor and the segment descriptor in which the global address occurs through the ELTA and SEGD fields, respectively. Defined symbol descriptors of one subprogram are linked to each other by the ELTL field. In addition, it is necessary that all defined symbols exist in one table. As this table is searched for specific symbolic addresses many times, it is best if the search takes as little time as possible. For this purpose, hash-coding is used. The F(DST) field contains the address of a table of addresses. A function  $f(s)$ , that uniformly distributes the set of all symbolic names  $s$ , is used to select one of the table locations,  $x$  ( $x \leftarrow F(DST) + f(s)$ ). This is called



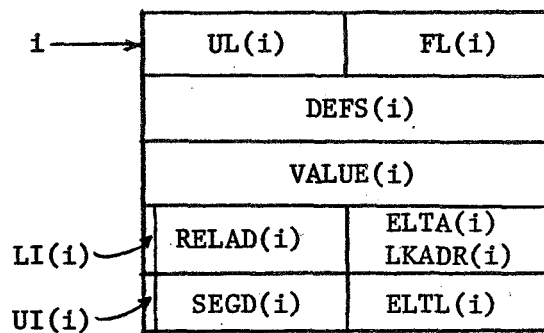
SEGE(i)	address of the segment descriptor
NEXT(i)	address of the next element descriptor
ELN(i)	symbolic name of the element
NB(i)	number of records of relocatable code on temporary storage
NC(i)	number of different common areas referenced in the subprogram
CSL(i)	address of the common symbol list for the element
NU(i)	number of different external references
USL(i)	address of the undefined symbol list for the element
NCA(i)	number of complex addresses in the subprogram
CAL(i)	address of the complex address list for the element
ND(i)	number of global addresses in the subprogram
DSL(i)	address of the chain of defined symbol descriptors for the element
SIZEL(i)	number of words required for the executable code of the subprogram
ABSLOC(i)	absolute address assigned to the subprogram
MASLOC(i)	mass storage location of the relocatable code

Figure 20 Element Descriptor



COMS(i)	symbolic name of the common area
MSIZ(i)	number of words in the common area
CABSLC(i)	absolute location assigned to the common area

Figure 21 Common Area Descriptor



UL(i)	address of the next defined symbol in the unresolved symbol table, or address of the next defined symbol in the overlay linkage table
FL(i)	address of the next defined symbol in the defined symbol table
DEFS(i)	symbolic name of the defined symbol
VALUE(i)	contents of the word corresponding to the defined symbol
LI(i)	library indicator bit
RELAD(i)	relative address of the defined symbol in the subprogram in which it is defined
SEGD(i)	address of the segment descriptor of the segment in which the defined symbol occurs
ELTL(i)	address of the next defined symbol in the defined symbol list of the element descriptor corresponding to the subprogram which contains the defined symbol
UI(i)	indicates whether or not the defined symbol has been resolved
ELTA(i)	address of element descriptor that corresponds to the subprogram that contains the defined symbol

Figure 22 Defined Symbol Descriptor

hash-coding and is further explained below. The chain of defined symbol descriptors pointed to by *x* contain symbolic names that produce the same value of *x* and are linked together by the FI field of each descriptor. Therefore, searching the defined symbol table for the descriptor with the same name, *s*, is just a matter of searching the subset of defined symbols that the function maps *s* into. Defined symbol descriptors are also placed into the table when no element has yet been found that contains them. In this case they are also linked to a chain of descriptors called the table of unresolved symbols. This table is pointed to by the UST table descriptor.

Figure 23 shows the format of the library queue entry. This is a two-word entry that is used to represent each library subprogram that must be input to complete the program. The queue is pointed to by the Q table descriptor. The LIFLOC field contains the mass storage location for the relocatable element containing the subprogram. The ESEG field contains the address of the segment descriptor to which the subprogram's element descriptor is to be attached. The queue of library entries are linked together by the NEXTQ field. The library queue is a chain of entries that varies in length. A chain of available entries is pointed to by the AVAIL table descriptor. These entries (shown in Figure 24) are linked together by the LINK field. As a new entry is needed, one is detached from the chain of available entries and attached to the queue. Once an entry has fulfilled its usefulness, it is returned to the chain.

Some of the inter-relationships between the descriptor tables are shown in Figure 25. The segment descriptors appear in the left-most column and are linked together via the NSEG fields. Each segment descriptor points to a chain of element descriptors in the center column. These descriptors are linked together within the chain. Each

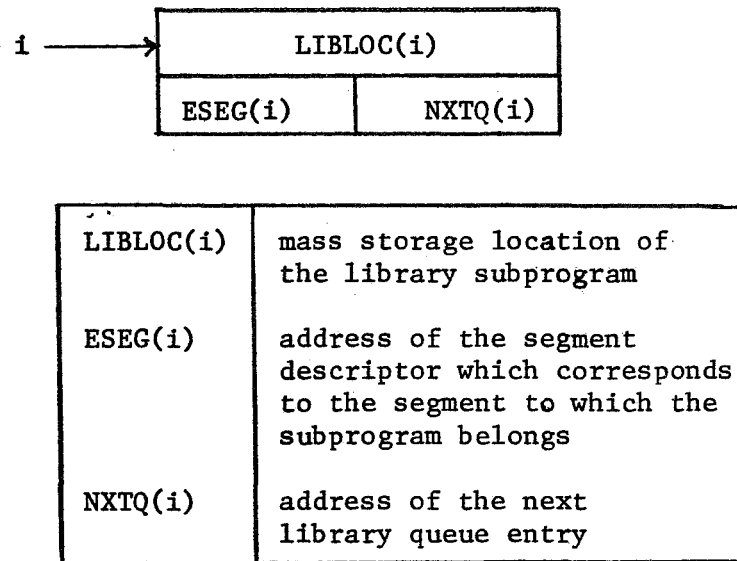


Figure 23 Library Queue Entry

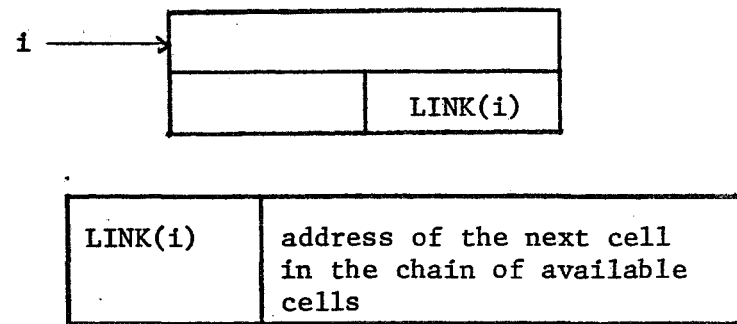


Figure 24 Chain of Available Cells Entry

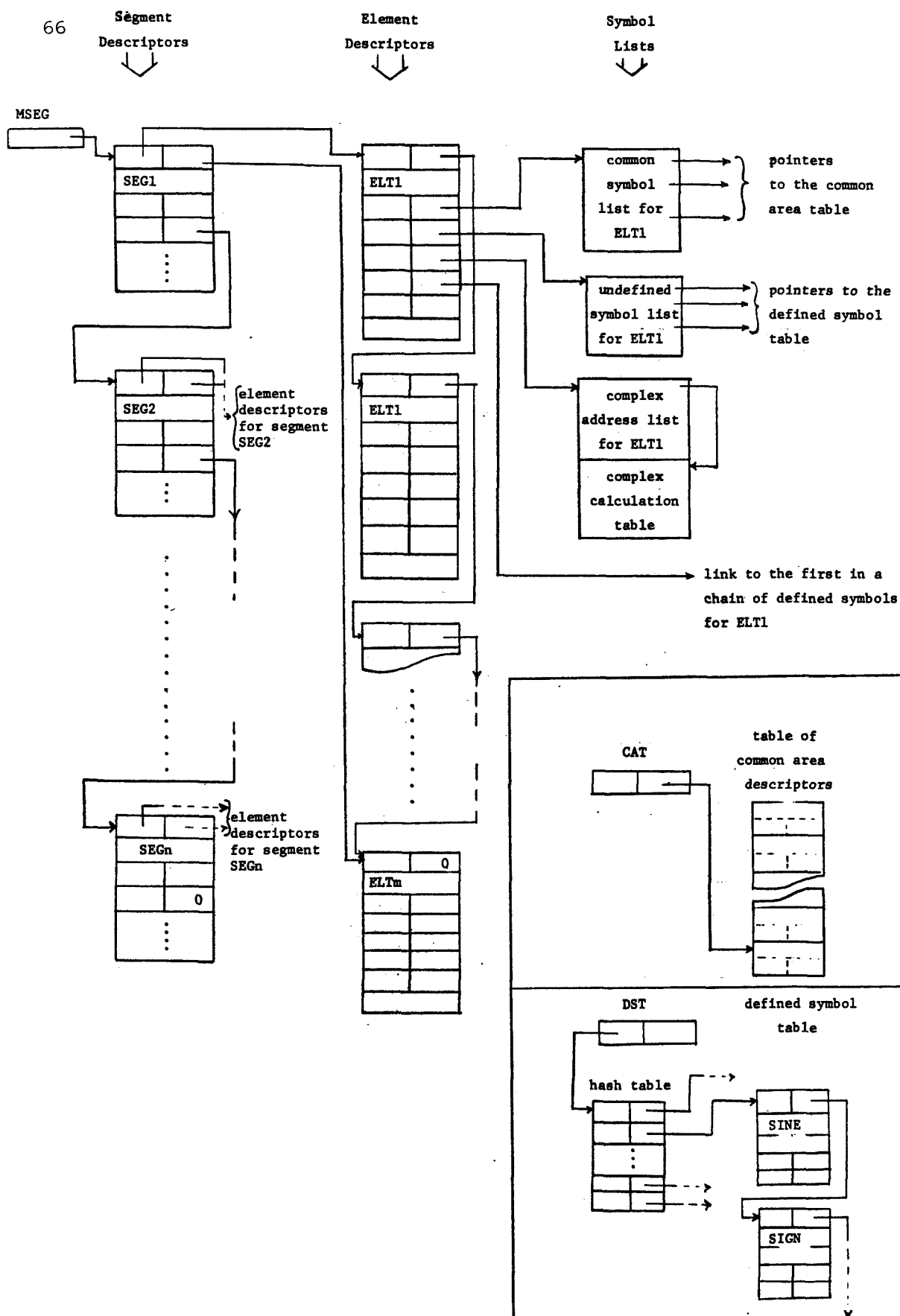


Figure 25

element descriptor contains pointers to four symbol lists for that element: the undefined symbol list, the defined symbol list, the common symbol list, and the complex address list. All except the defined symbol list are stored sequentially. The defined symbol list for each element consists of a series of chains of defined symbols that are referenced through a hash table. This example does not show all the possible relationships due to the complexity of the resulting diagram. For example, each element descriptor points back to its corresponding segment descriptor. For this reason the examples presented in the following sections will instead show a "snapshot" of memory, with the actual addresses given.

Several conditions led to the data structure described above. First, the number of descriptors of any one type is not known a priori. By allowing all the tables to exist within the same area, maximum use is made of available space without the bother of repositioning tables. Second, certain descriptors had to exist within several tables. For example, the defined symbol descriptor entry is linked to the defined symbol table and to the defined symbol list for the element in which it occurred, and possibly to the UST or FIT chains. Third, both the segment descriptor table and the element descriptors require re-ordering. With sequential lists re-ordering is difficult but with linked lists it is relatively easy.

In regard to the defined symbol table, it is worthwhile to discuss alternative methods of organization. The algorithm requires that the table be searched frequently and that additions be made to the table when any search is unsuccessful. The solutions considered were:

- (a) Linear\_search . Link all symbols together in a linear list.
- (b) Binary\_search . Assign an order to all symbols.


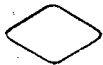




Link all symbols together in a binary tree, where the left-link always points to a symbol of lower order, and the right-link always points to a symbol of higher order.

- (c) Hash-coding . Map a function  $f(s)$ , on the symbolic name  $s$ , to a table of addresses, where each address points to a chain of symbols with identical values of  $f(s)$ .

The first method, the linear list is the simplest method to implement but its operation slows down linearly with the number of entries. The second method, the binary search, offers the fastest search of a linked list that can be made. However, the method requires that the binary tree being searched is balanced. As new entries are attached to the tree the necessity of rebalancing arises. As the list of defined symbols is constantly expanding, the overhead in this method is too great. The method chosen, hash-coding, constrains the search to a subset of all entries. The larger the hash table the faster the search. In order to achieve maximum desirability, the hash function should map the global symbols to the table in a uniform distribution, and the table should not be so large as to waste space, but large enough so that the search does not require an inordinate amount of time. An example would be a function that mapped the first character of the name to a table of 64 entries (considering a 6-bit character). The actual size of the table and definition of  $f(s)$  will depend upon the particular implementation. For this reason, neither is specified within this report. (See [15] for a survey of hash-coding techniques.)

## 4.2 Detailed Flow Charts

The algorithm is presented in this section as a series of detailed flow charts with accompanying description. The format of the flow charts is fairly simple. The backward arrow ( $\leftarrow$  or  $\leftarrow$ ) is used as a replacement operator. The relational operators ( $=, \neq, <, \leq, >, \geq$ ) are only used at a program branch point. The box () is used to enclose replacement operations. The diamond () and the hexagon () are used to denote program branch points. The oval () is used to denote a function call. All functions that are called, except SEC or IO, have a corresponding flow chart defining them. Four functions, CH(i,j), BYT(i,j,k), f(s), and g(s) are used in the replacement operations and are defined at their first appearance.

At several points throughout the algorithm, tests are made for an error condition (e.g., table overflow, improper format, etc.). These error conditions may be fatal or non-fatal. Non-fatal errors are noted within the flow chart. This replaces the typical action of a diagnostic message to the user. Fatal errors cause a branch to an error exit, labelled E. Rather than present an error-handling routine, a table of fatal error types is shown in Table 1.

### 4.2.1 Auxiliary Routines

In order to concentrate on the algorithm for a loader the details of input, output, and mass storage will be avoided. Two functions, IC and SEC, will be defined to perform these tasks. The function IO performs the set of

E1	I/O error condition from the function IO
E2	file system error from the function SEC
E3	illegal control record encountered in user input
E4	segment identifier missing from \$SEG control record
E5	available table space (TBL) overflowed
E6	more than one main subprogram is specified
E7	a segment is referenced but is no where defined
E8	the partial ordering given in the overlay description is not complete
E9	an element descriptor is referenced but is nowhere defined
E10	a portion of the library index is referenced but cannot be found
E11	the memory space available for program allocation has been exceeded
E12	the mass storage location referenced contains no information
E13	an address reference is made between segments that overlay each other
E14	the list of available cells is empty

Table 1. Fatal error conditions for the loader algorithm

input/output operations with a specified unit. The five operations used within the algorithm are detailed in Table 2(a).

The function SEC is a simple file maintenance system that will be referred to as the 'secretary.' The five operations that the secretary performs are detailed in Table 2(b). The format of the function call is SEC (i, MNAME, . . .), where i specifies the operation and MNAME is the file identifier. The file identifier is a symbolic name associated with a given set of records, termed a file. The first operation (i=1) reads a record from the file MNAME. The second operation (i=2) writes a record to the file MNAME. In both operations the file is positioned at the next record once the operation is completed. The third operation (i=3) assigns a symbolic file name to MNAME. The fourth operation (i=4) positions the file to the specified record in the file MNAME. The fifth operation (i=5) releases the file MNAME to the secretary. The file acts like a separate I/O unit. Successive read operations will read successive records. Therefore, in the case of one record to a file, the secretary provides symbolic random access to information placed on mass storage whether or not actual random access is possible.

Although not shown in the table, both functions provide three return conditions when their tasks are complete, normal (N), error (ERR), and end-of-information (ECI). The normal return occurs when the tasks are completed as requested. An error return indicates an input/output unit error, while an end-of-file return indicates that a physical end-of-file mark was encountered during an operation of the IO function or that an end-of-information was encountered in the function SEC.

Three other routines are detailed in this section, BUILDLIST, PUT, and GET. These three routines, described as

## (a) IO: the I/O routines

IO(1,IU,PLACE,N)	read a record from unit IU, transmit N words of it to core location PLACE and following
IO(2,IU,PLACE,N)	write a block of N words as a record to unit IU from core location PLACE through PLACE+N-1
IO(3,IU,PLACE,N)	read a record from unit IU to core locations PLACE and following, place the number of words in N
IO(4,IU)	rewind IU
IO(5,IU)	write an end-of-file on IU

## (b) SEC: the I/O secretary

SEC(1,MNAME,PLACE,N)	read a record from mass storage file MNAME to core locations PLACE and following; place the number of words in N
SEC(2,MNAME,PLACE,N)	write a record of N words from core locations PLACE through PLACE+N-1 to mass storage file MNAME
SEC(3,MNAME)	provide a file identifier in MNAME
SEC(4,MNAME,I)	position to the I-th record of mass storage file MNAME
SEC(5,MNAME)	release mass storage file MNAME

Table 2 Description of the function calls to the I/O routines and the I/O secretary

flow charts in Figure 26, create and provide access to the list of available cells. BUILDLIST creates the chain of cells in the area of the output buffer. F(AVAIL) points to the first cell in this chain. The function call PUT (i) places a two-word cell at address i back onto the chain. The function call GET (i) detaches a two-word cell from the chain and places the address of the cell in i. These two-word cells are used to form the library subprogram queue.

#### 4.2.2 Input And Table Construction

The first phase of the loader algorithm assembles all the input, overlay description and relocatable elements, into a data structure that represents the program. Figure 27 shows the general flow of control for this phase. Once initialized, the task consists of:

- (a) processing the user input,
- (b) ordering the segment descriptors if overlay was indicated,
- (c) cross-referencing the undefined and defined symbol tables,
- (d) resolving the symbolic addresses left unresolved by step (c) in the library index,
- (e) inputting each relocatable element from the library and cross-referencing its undefined symbols with the defined symbol table,
- (f) repeating steps (d) and (e) until the program is complete.

The details of the steps above are described in the detailed flow charts which follow.

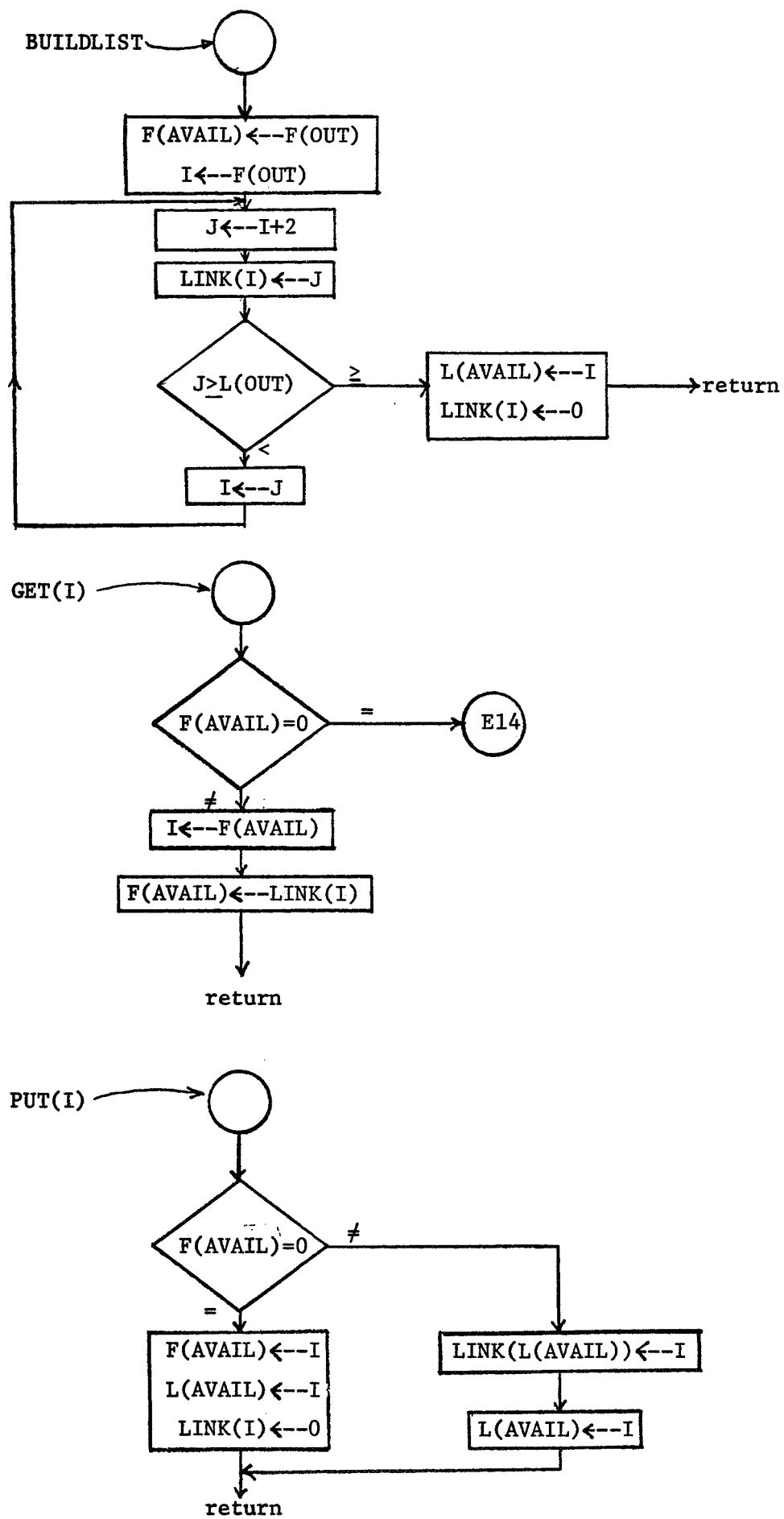


Figure 26

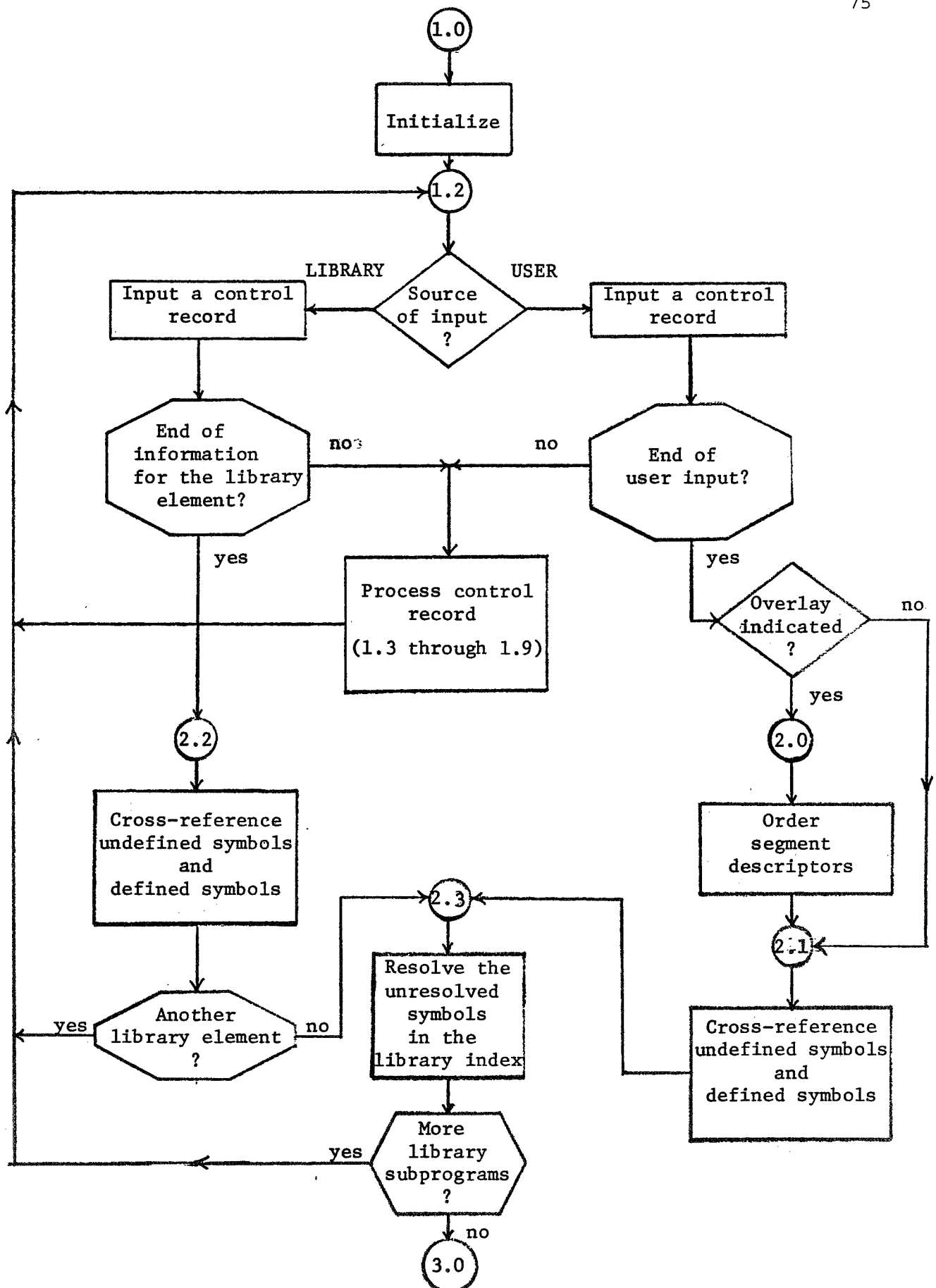


Figure 27 Flow chart Showing Input and Table Construction



### Initialization

Initialization is shown in Figure 28, and comprises four steps. First, an initial segment descriptor is set up in case the user does not require overlay, allowing his program to be treated as a single segment. Second, the common area table is initialized at the high address end of the table TBL. Third, the mode of the input is set to indicate user input. Fourth, a list of two-word cells (AVAIL) is created in the output buffer by the function BUILDLIST. It should be noted that S is used as a pointer to the next available location in the table TBL. It shall retain this identity throughout the first phase.

### Control Record Input And Interpretation

Monitoring of the control record input and interpretation is shown in Figure 29. In the user mode, the control record is input from the I/O unit INU to the input buffer which begins at F(INP). In the library mode, the control record is input from the mass storage location indicated by the first element in the library queue (F(Q)). In either case, the control record is processed in one of seven ways, depending upon the control word. At this point the next control record is input. If an end of information (EOI) is encountered in the user mode, then the user input is exhausted. At this point, a test is made to see if more than one segment is defined. If so, then overlay is indicated and the overlay segment loader must be input from the library. This is accomplished by entering the mass storage location of \$LINK\$ (stored as variable LINKER) into the library queue. In addition, the segment descriptors must be re-ordered (see Figure 39). If there is only one segment, this segment is defined as the main segment, and the loader cross-references the undefined symbol list with the defined symbol table (see Figure 44). An end-of-information in the library mode signals the completion of

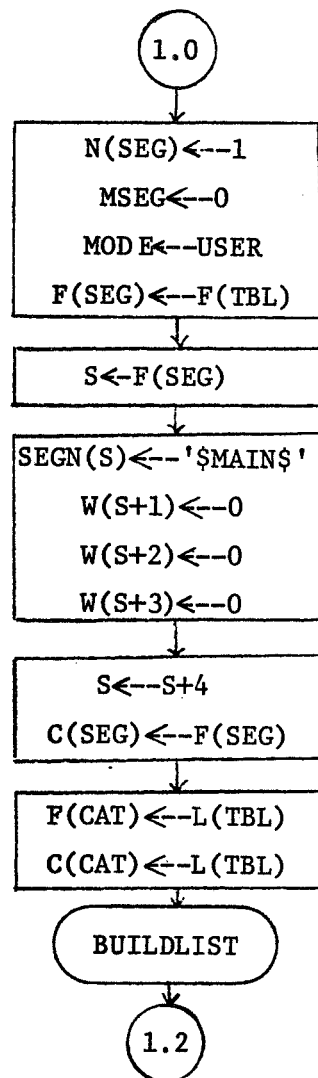


Figure 28

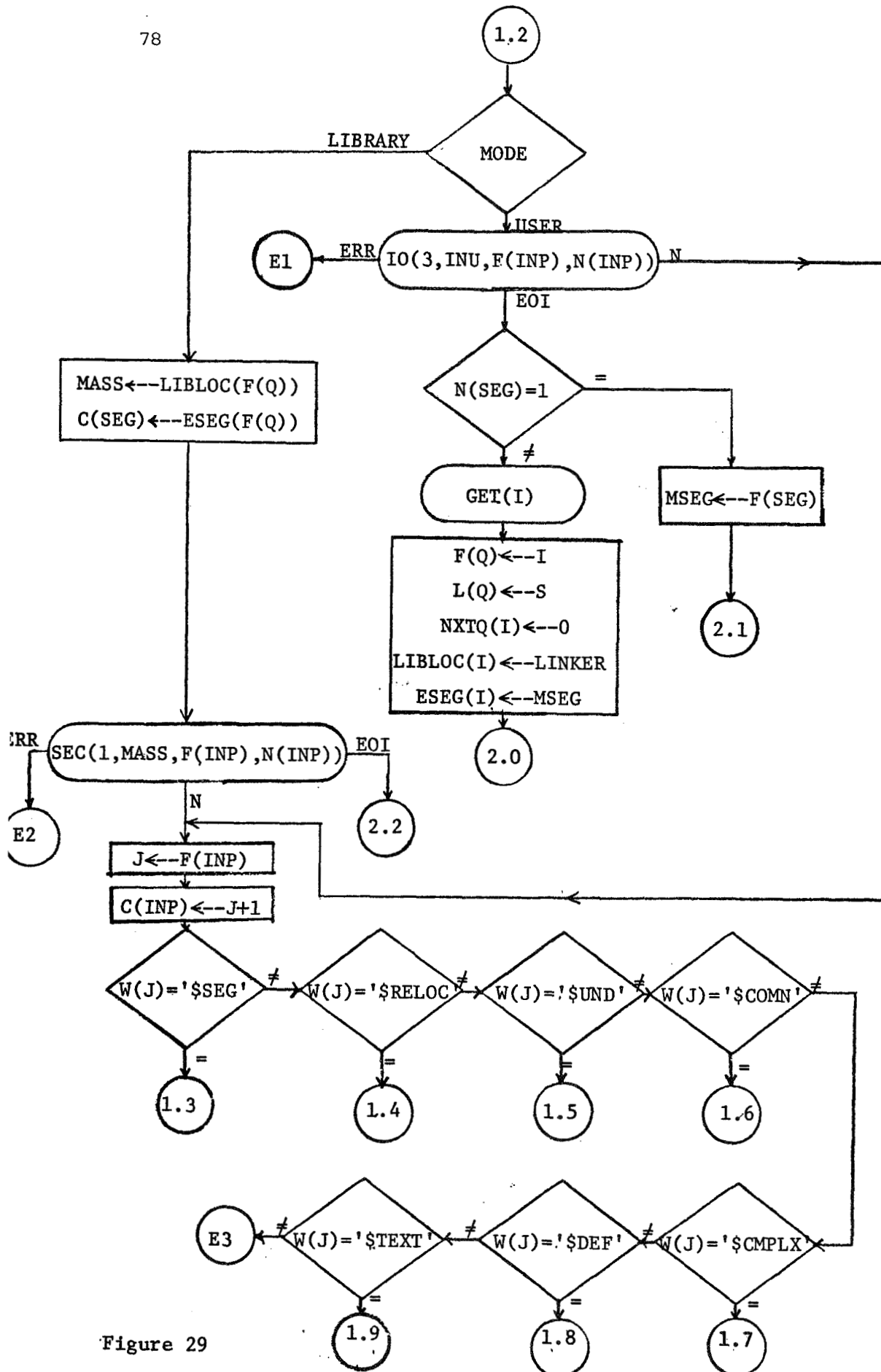


Figure 29

input for the library subprogram indicated by the first cell in the library queue. At this point, the loader cross-references the subprogram's undefined symbol list with the defined symbol table (see Figure 44).

Figures 30 through 37 show the processing of the seven types of control records. The \$SEG control record (Figures 30 and 31) is processed lexically to obtain the symbolic name assigned to the segment and the symbolic names of its predecessor segments. The lexical processing employs a function CH(i,j) which extracts the j-th character from the string of characters beginning at i. In the flow charts, 'b' is used to represent a blank character. A segment descriptor is created for the segment at location S. If this is the first segment descriptor, the variable MSEG is loaded with its address. The symbolic name of the segment is placed in the SEGS field of the descriptor, and the names of the predecessors are placed in a list at the end of the descriptor. The number of predecessors is placed in the MPEI field of the descriptor. The new segment descriptor is linked to the previous segment descriptor through the NXTSEQ field of the previous segment descriptor.

The \$RELOC control record (Figure 32) causes an element descriptor to be created in the table area TBL. This descriptor is added to the chain of element descriptors specified by the STOP and SBOT fields of the current segment descriptor. The link is made through the NEXT field of the previous element descriptor. The fields are initialized at zero except for the EIN field which is loaded with the name of the element, the SIZEI field which is loaded with the size of the element, and the SEGE field which is loaded with the address of the current segment descriptor. The first address field of the third word of the \$RELOC control record indicates whether or not the subprogram contains the program starting location. If it does, then the address of the

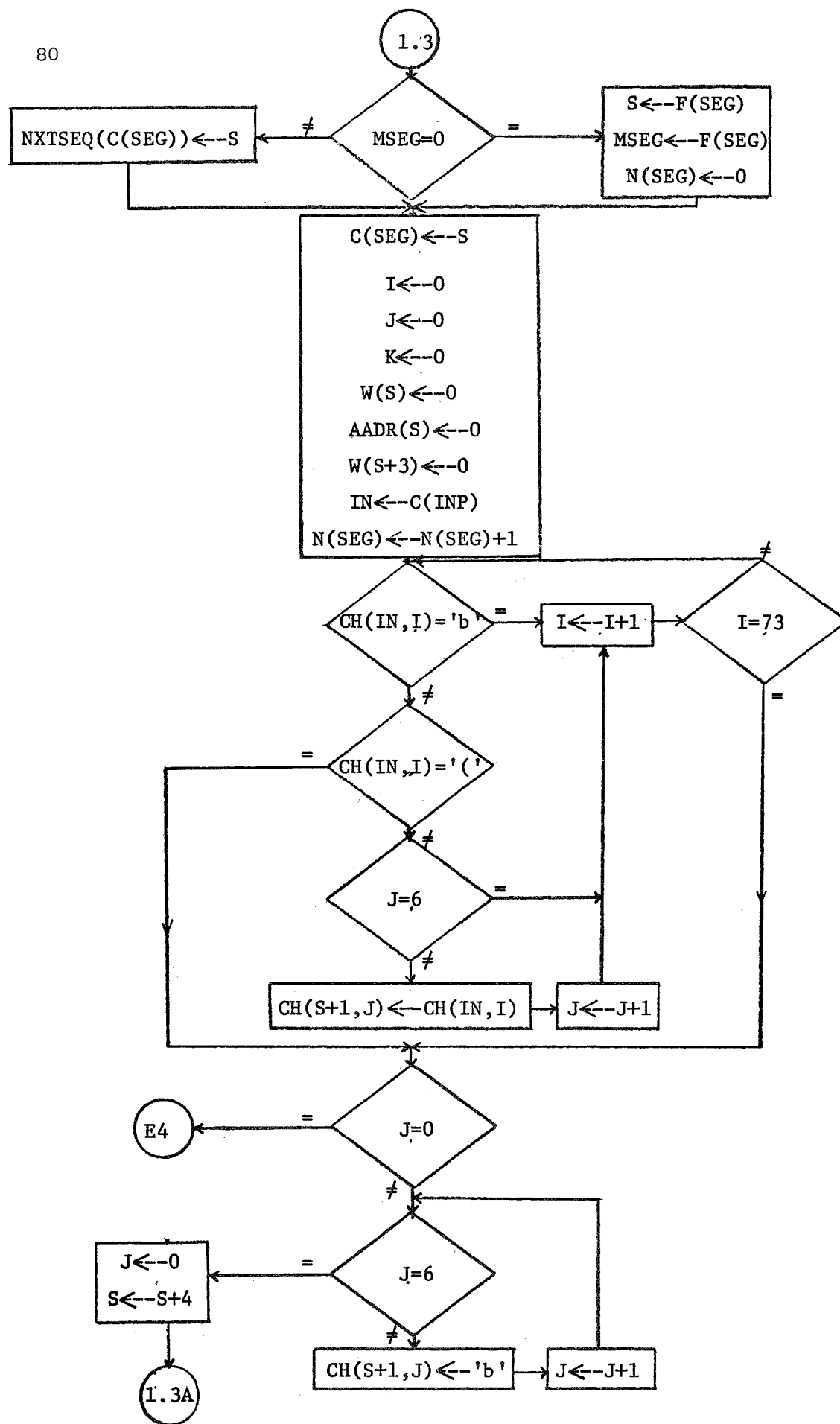


Figure 30

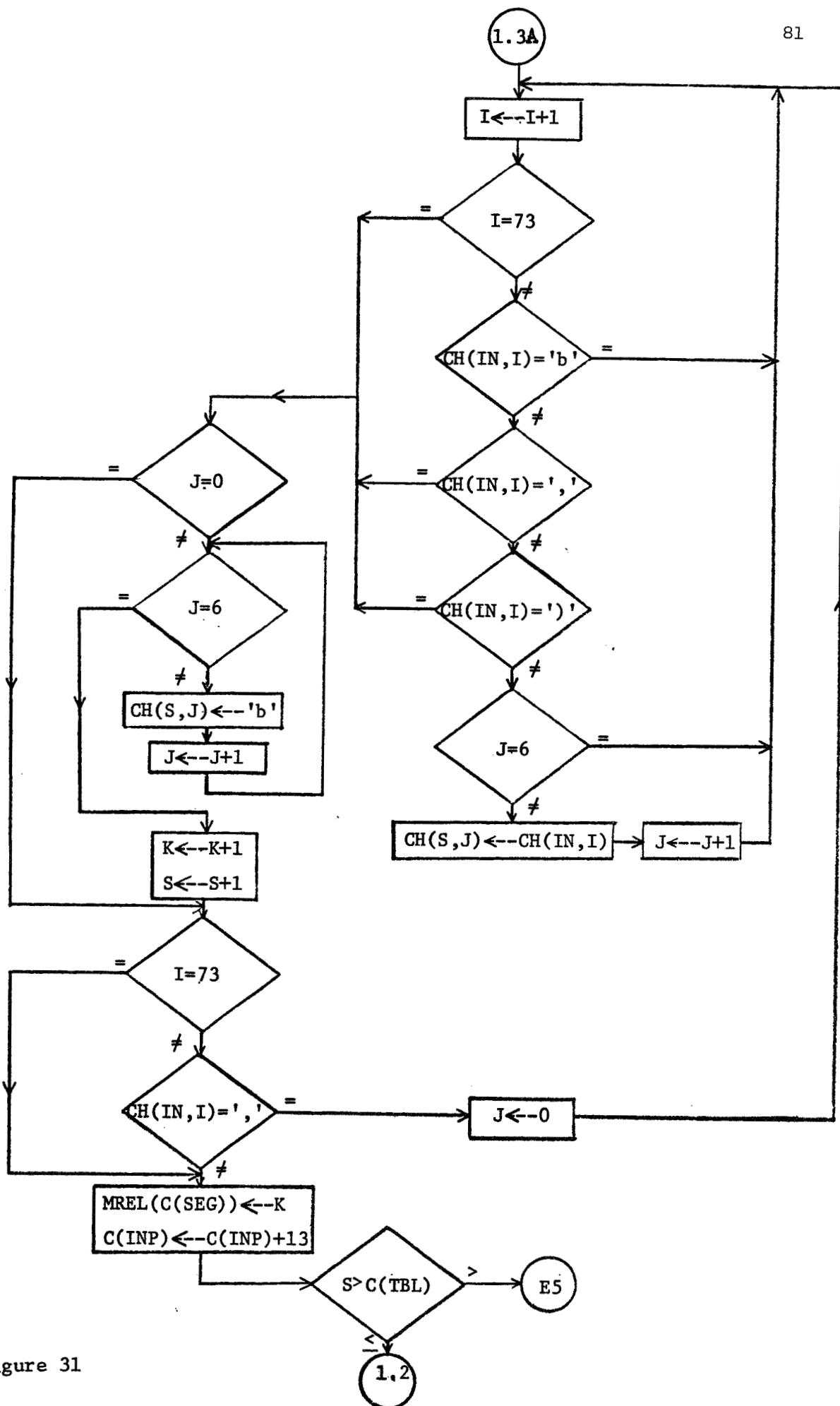


Figure 31

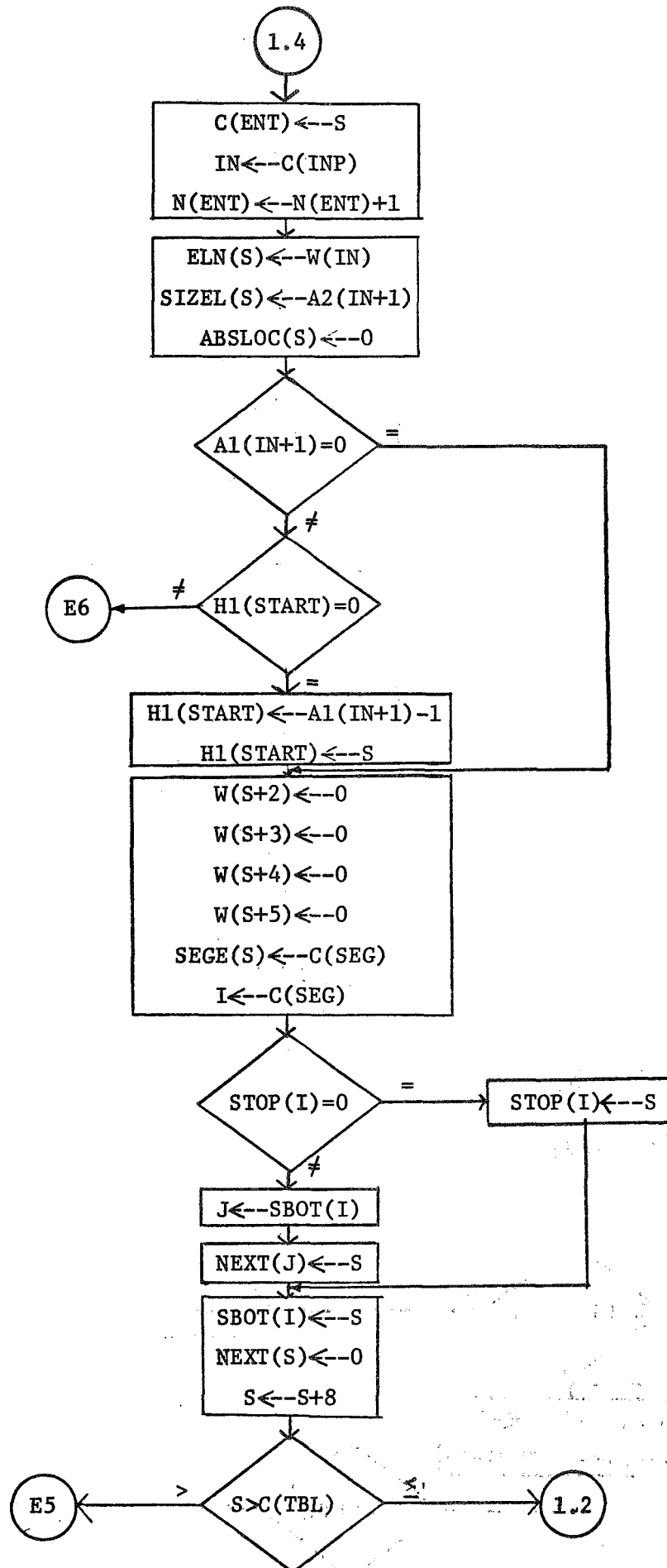


Figure 32

program starting location is placed in the first half of variable START, and the address of the element descriptor is placed in the second half.

The \$UND control record (Figure 33) causes the list of undefined symbols to be added to the current element descriptor. The list of undefined symbolic addresses is added to the table area TBL, the address of the list is placed in the USI field of the current element descriptor, and the number of undefined symbolic entries is placed in the NU field.

The \$CCMN control record (Figure 34) requires several steps. First, for each entry in the control word a one word entry must be added to the element descriptor; the element descriptor field CSI must point to this block of entries; and field NC must contain the number of entries in the block. Second, for each entry in the control record that does not have a matching descriptor in the common area table (CAT) (matching means the same symbolic name), a new common area descriptor must be created. Third, the common symbol list entries in the element descriptor must be linked to (i.e., loaded with the addresses of) the corresponding entries in the common area table. Finally, it must be decided whether the common area is defined or undefined in this subprogram. If the common area has already been identified as defined in another subprogram (MSIZ=0), the common area table entry is left as is. If the common area is defined in this element, then the pointer to it in the element descriptor is flagged by setting the first half of the pointer word to 1. Otherwise, the common area is undefined, and the field MSIZ in the common area descriptor is adjusted to correspond to the larger of the sizes appearing in the control record and the common area descriptor.



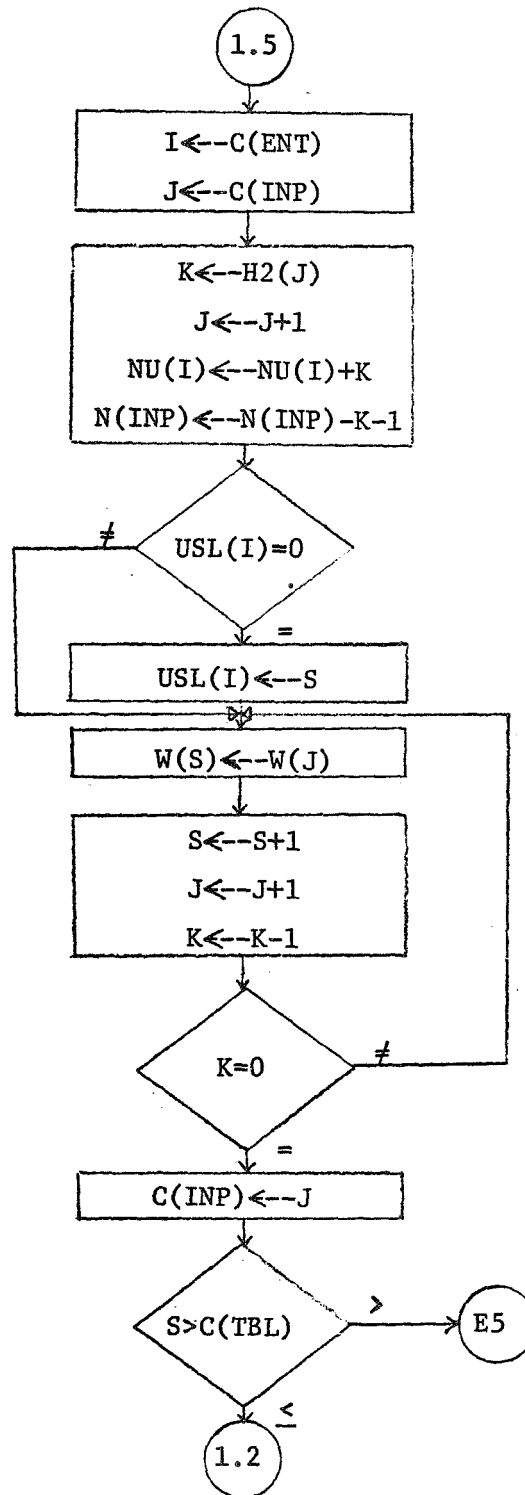


Figure 33

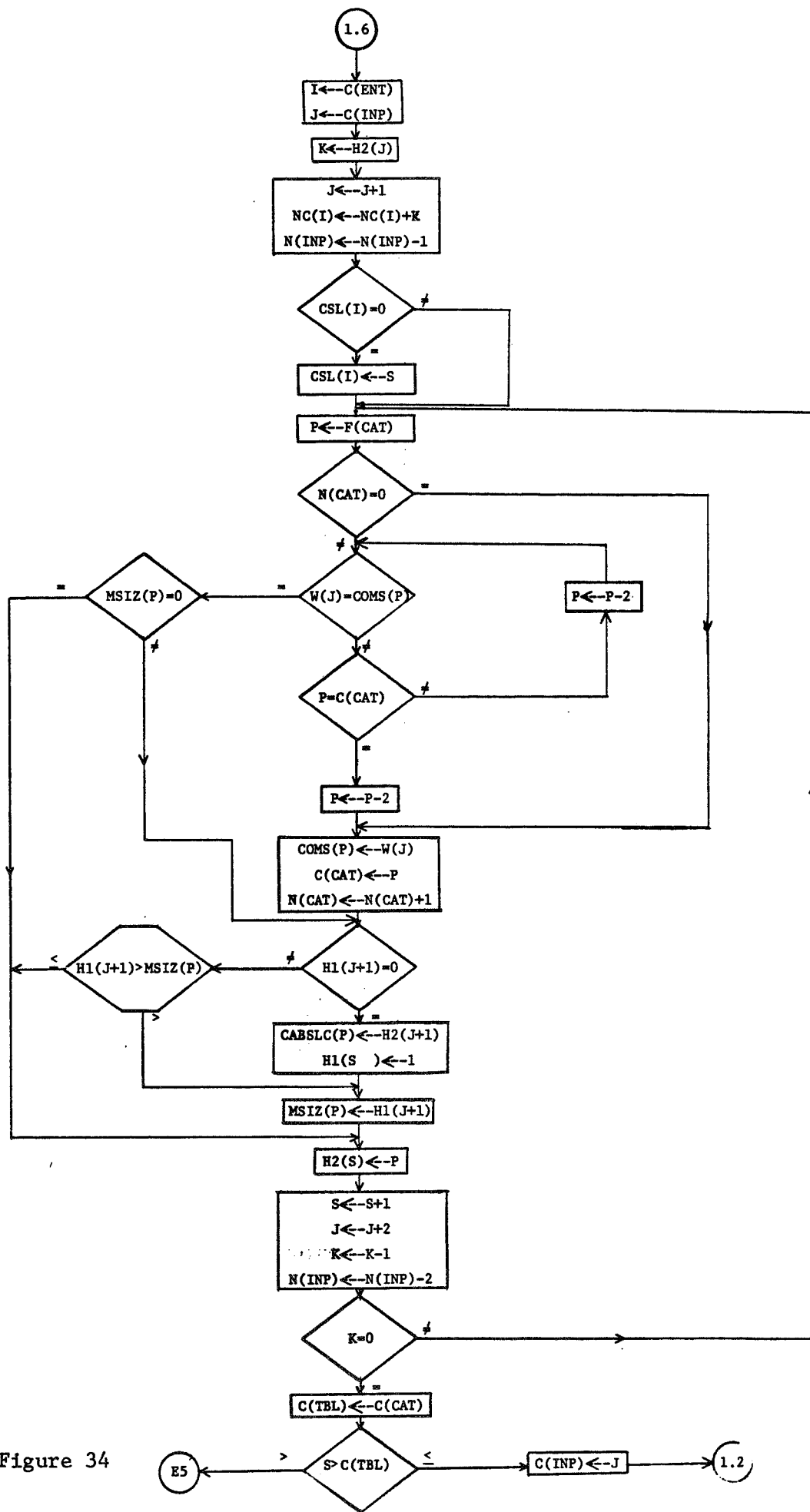


Figure 34

The \$CMFLX control record (Figure 35) is processed in four steps. In order that there is a place to store the results of the complex calculations, a block of empty words equal to the number of complex addresses indicated in the control record is added to the element descriptor. The NCA and CAI fields of the element descriptor are loaded with the number of words and address of the block, respectively. The contents of the control record (the complex calculation instructions) are added to the element descriptor as a block of words. The first word of the empty block is loaded with the address of this block.

The \$DEF control record (Figure 36) is processed in four steps, an entry at a time. First, the defined symbol table is searched for a previous entry of the same symbolic address. As this search is made frequently, the table is stored as several chains of symbolic addresses. A hash-code function  $f(s)$ , where  $s$  is a symbolic address, is used to determine in which chain a particular symbolic address appears. The EST entry points the hash table. Second, if the entry does not exist in the chain specified by the hash-code function, then a new entry is made at the beginning of the chain and the symbolic address is loaded into the DEFS field of the entry. If the entry does already exist, one of two conditions exist. Either the symbolic name is being defined twice, which is an error condition, or the entry was made during defined symbol-undefined symbol resolution and must be completed. In the latter case, the UI field of the entry is equal to 1, and the defined symbol entry is added to the element's chain of defined symbols. This chain is addressed through the DSL field of the element descriptor and is linked together through the ELIL field of the defined symbol entries. In this manner, the entry is linked to two chains, the other being the defined symbol table chain. Fourth, the RELAC, IKADR, and VALUE fields are loaded.

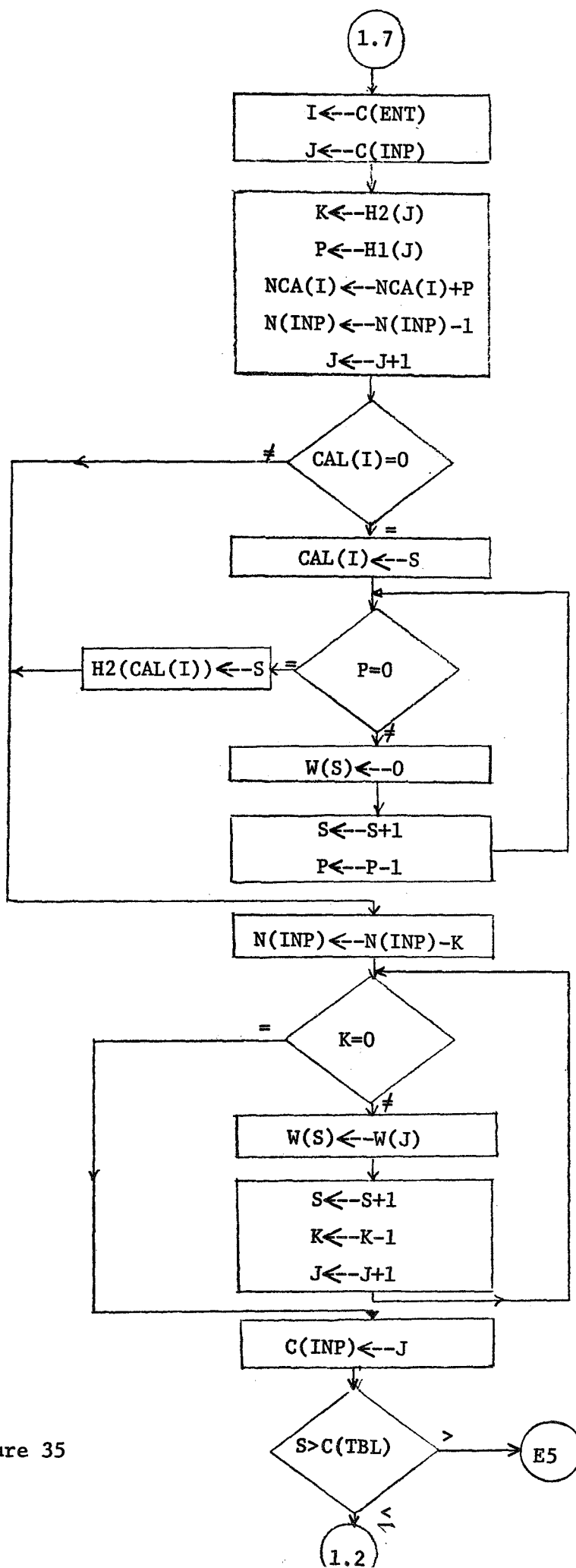


Figure 35

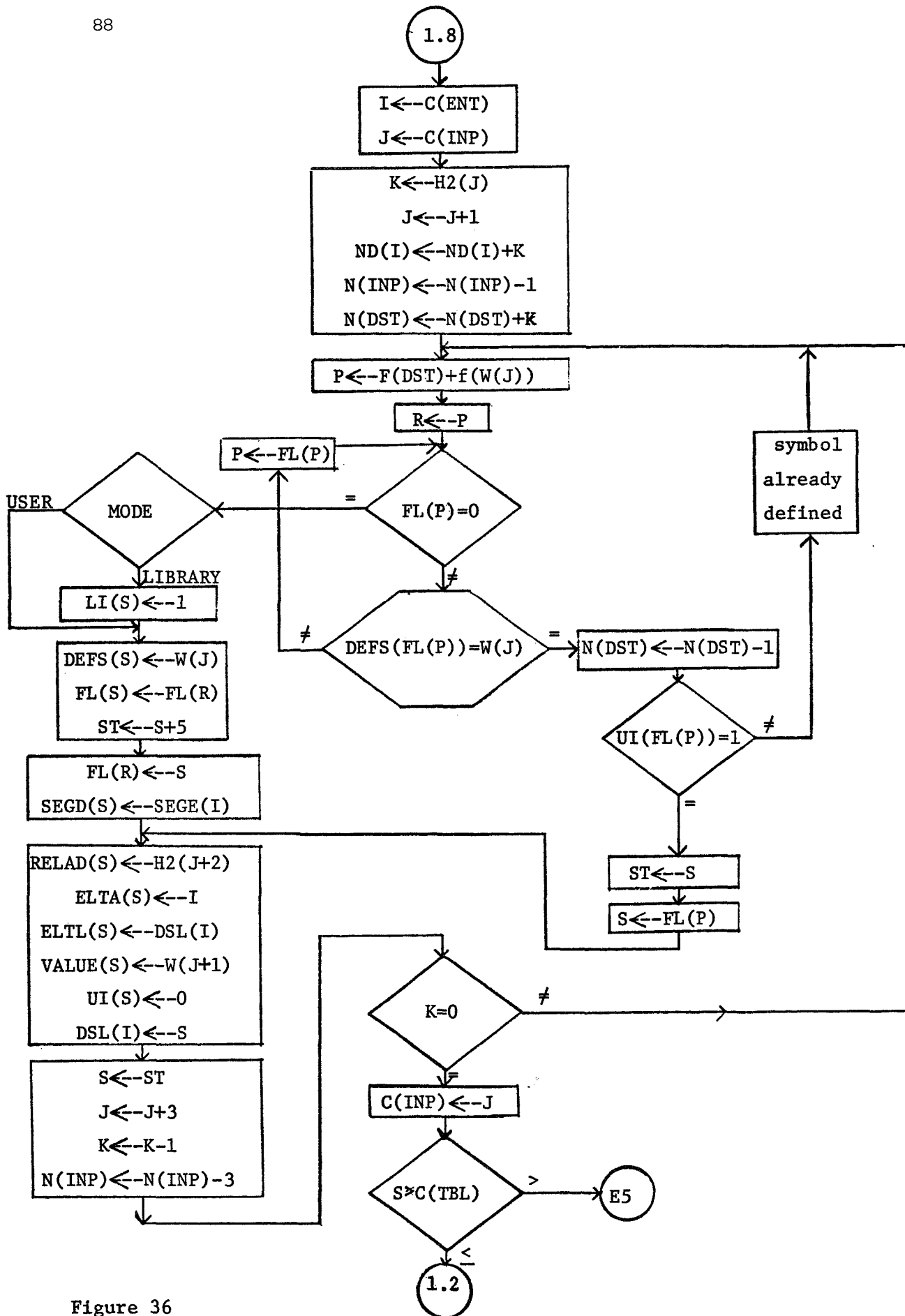


Figure 36

The \$TEXT control records (Figure 37) contain the relocatable code for the subprogram. These records are stored temporarily onto mass storage through the secretary. The secretary provides an address which is stored in the MASICC field of the element descriptor. All of the \$TEXT control records for one element are written to the same address.

As an example, Figure 38 shows part of the memory contents after processing the user input shown in Figure 7. The table area TBL extends from 1000<sub>8</sub> to 7777<sub>8</sub>. The output buffer extends from 11000<sub>8</sub> to 11777<sub>8</sub>. The segment descriptor is in locations 1000<sub>8</sub> through 1004<sub>8</sub> and indicates that there is one segment (i.e., the NXTSEQ field contains a zero) and one element (SBOT(1000<sub>8</sub>)=STOP(1000<sub>8</sub>)=1005<sub>8</sub>). The element descriptor begins at 1005<sub>8</sub> and extends through 1020 and is translated as follows. The element's symbolic name is MAIN. The element contains one common area (NC(1005<sub>8</sub>)=1) the address of which is at 1020<sub>8</sub> and three undefined symbols (NU(1005<sub>8</sub>)=3), the symbolic addresses of which begin at 1015<sub>8</sub>. The executable code, when translated from the relocatable code (stored at MASICC(1005<sub>8</sub>)=TEXT01), will require 12 words of memory. The three undefined symbolic addresses are SIN, CCS, and SUBR. The common area descriptor can be found at location 7776<sub>8</sub>.

The common area table in the example consists of one entry at the other end of the table area TBL (address 7776<sub>8</sub>). This entry is for the common area BLOCK and indicates that the common area must have 10<sub>8</sub> memory words when allocated.

The output buffer in the example (words 11000<sub>8</sub> to 11777<sub>8</sub>) contains the chain of available cells. The second word of each two-word cell contains the address of the next cell in the chain. The F field of the pointer entry AVAIL points to the first cell in the chain, while the entry

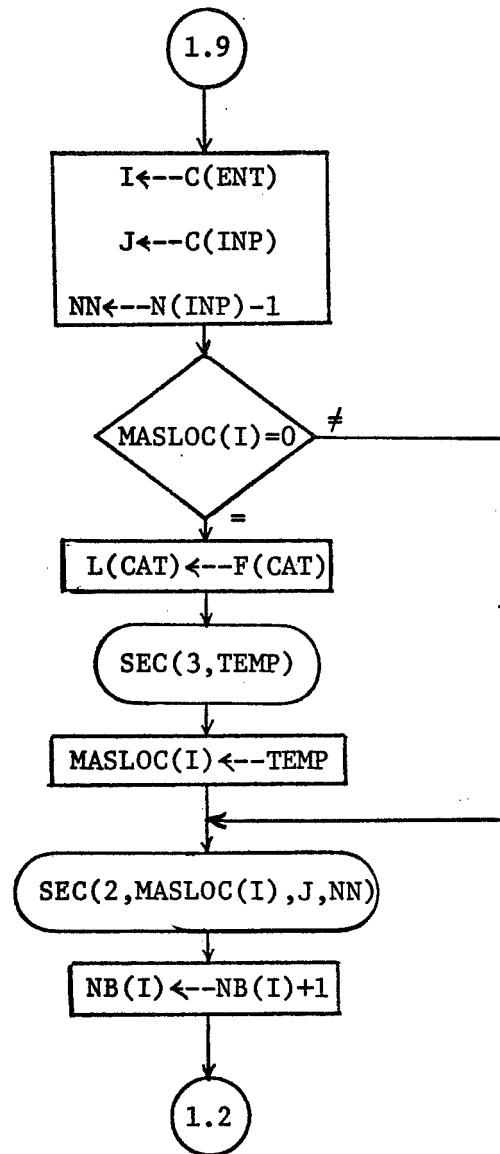


Figure 37





L(AVAIL) points to the last cell in the chain.

### Re-ordering Of Segment Descriptors

Once the user input is exhausted, if memory overlay has been indicated (NSEG#1), then the segment descriptors are re-ordered. Each \$SEG control record contains a list of symbolic names of the segment's immediate predecessors in the overlay structure. Examination of an overlay structure shows that the segments are partially ordered by the segment specification. A set of segments is partially ordered if a relationship  $\leq$  exists between them such that, for segments  $x$ ,  $y$ , and  $z$ :

- (a) if  $x \leq y$  and  $y \leq z$ , then  $x \leq z$ ;
- (b) if  $x \leq y$  and  $y \leq x$ , then  $x=y$ ;
- (c)  $x \leq x$ .

The relation  $x \leq y$  is read "'x precedes or is equal to y.'" In the segment specification, each predecessor (PREDi) listed with the segment identifier (SNAME) would have the relation  $PREDi \leq SNAME$ . As the predecessor list does not allow equalities to be shown, conditions (a) and (b) are met by the linearity of the memory and condition (c) is understood but never specified.

It is necessary to sort the segments in such a way that no segment is allocated memory prior to a predecessor segment because to allocate memory space to a segment requires the location of all predecessor segments. D. E. Knuth [14] provides an algorithm for this very problem. Given an input of relationships that form a partial ordering in a set of elements, the algorithm performs a topological sort (i.e., it embeds the partial order in a linear order). The algorithm is modified slightly to suit the data structure of the loader.

The topological sort is described in Figure 39. The sort requires that each segment to be ordered has a list of the segments that are specified as following it in the

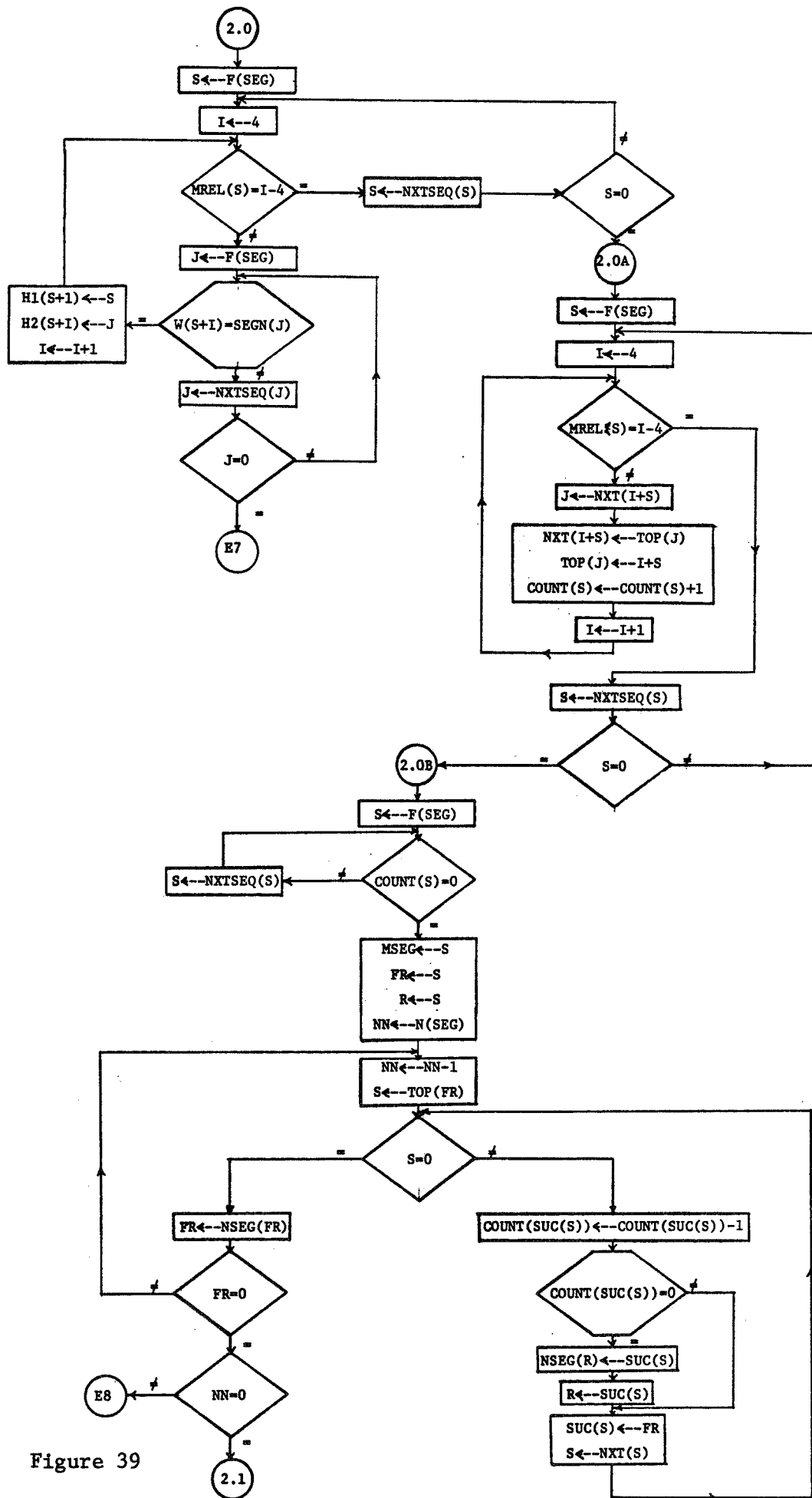


Figure 39

partial order. In other words, a segment SNAME would have a list of all the segment identifiers corresponding to predecessor lists in which SNAME appears. This list is called the list of direct successors. However, the data structure associates each segment with its list of direct predecessors. Therefore, the first task of the sort is to prepare the structure.

Between 2.0 and 2.0A in the flow chart in Figure 39, each entry in the list of symbolic segment identifiers at the end of each segment descriptor is changed to a word containing two pointers. The first pointer contains the address of the segment descriptor to which the list is attached, while the second pointer contains the address of the segment descriptor previously named in the entry.

Between 2.0A and 2.0E in the flow chart, the list of predecessor relation words is processed again. Each entry is now added to the chain of direct successors of the segment descriptor that is pointed to in the second half of the word. To do this the TOP field of the segment descriptor is loaded with the address of the word that pointed to the segment descriptor, and the second half of the word is replaced with the old contents of the TOP field. In addition, the CCOUNT field of each segment descriptor is loaded with the number of relation words that appear in the list at the end of the segment descriptor. The COUNT field contains the number of direct predecessors of the segment, and the TOP field contains the address of the first in a chain of one word direct successor cells. Each cell has a SUC field, that contains the address of the segment descriptor that it represents, and a NXT field, that contains the address of the next cell in the chain.

After 2.0E in the flow chart the topological sort begins. The sort is initialized by finding the segment descriptor with a zero CCOUNT field, indicating no

predecessors. There should be only one such segment, and this is called the "main segment." The variable MSEG is loaded with the address of this segment descriptor. The sort begins at this point. The list of direct successors is processed. As each cell is encountered, the CCUNT field of the segment to which the cell corresponds (addressed by the SUC field) is decremented. If the CCUNT field reaches zero, the segment descriptor is added to the chain of ordered segment descriptors (linked through the NSEQ field). In addition, the cell is removed from the chain and once again loaded with the address of the predecessor. The variable PR points at the segment descriptor whose successor chain is currently being processed, while the variable R points to the end of the chain of ordered segment descriptors. The algorithm terminates when the CCUNT field of all segments has gone to zero. If the variable NN is not zero, then all segment descriptors have not been added to the chain of ordered segment descriptors. This can only mean that there is a disjoint set of segments in the overlay description. This must be treated as a fatal error.

Figures 40 through 43 show the change in the segment descriptors of the example shown in Figure 10 during the sort. Figure 40 shows the eleven segment descriptors with the symbolic name in the second word of each descriptor and with the list of predecessors beginning in the fifth word of each cell. The segment descriptors are each assigned an address and linked together via the NXTSEQ field in the order in which they were input (shown by arrows also). Figure 41 shows the segment descriptors after the first step. It should be noted that each predecessor entry contains the address of the segment descriptor in which it appears in addition to the pointer to the predecessor. Figure 42 shows the descriptors after the second step. The TCF field of each descriptor points to a chain of successors. For example, the successors of MAIN are B, A,

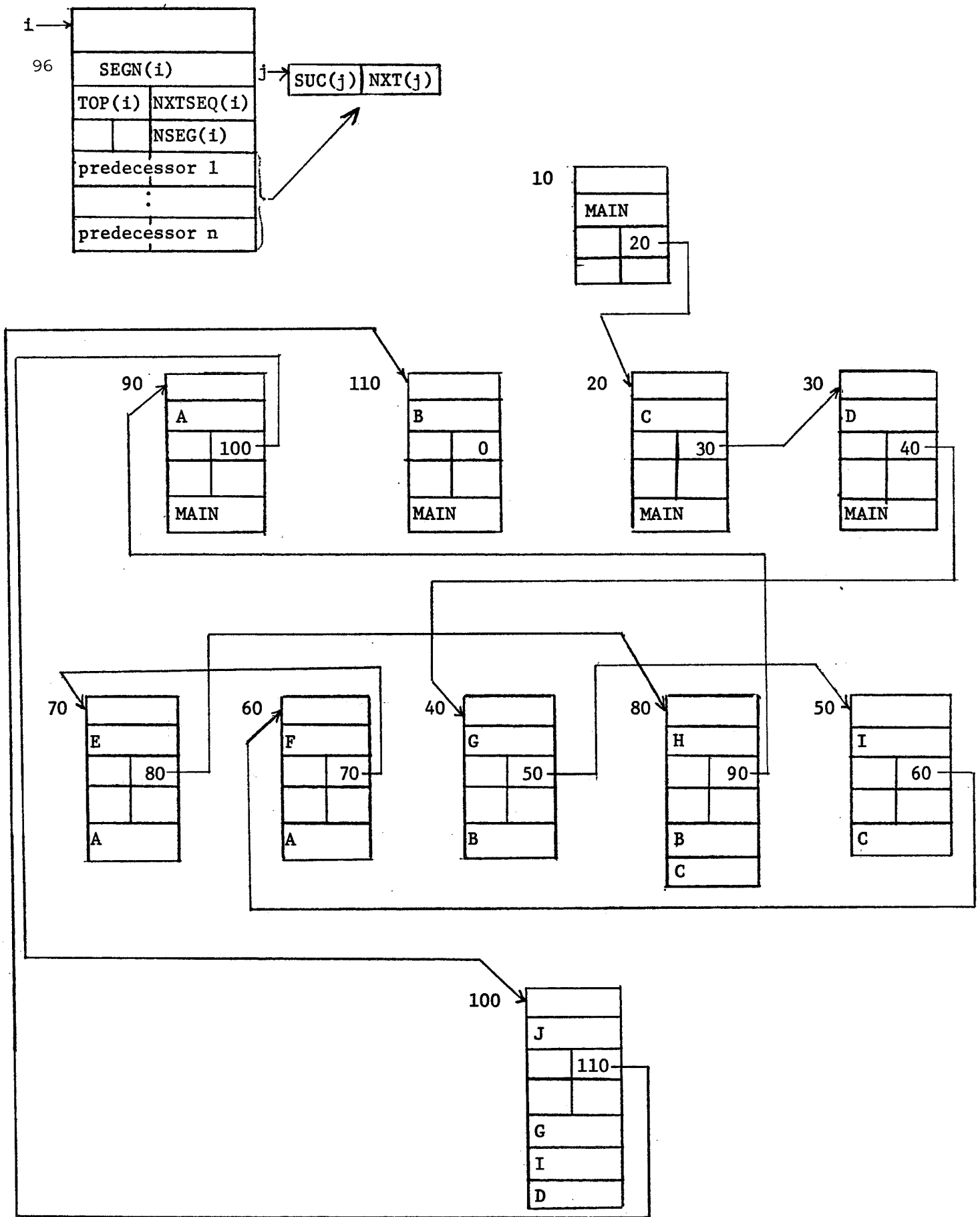


Figure 40

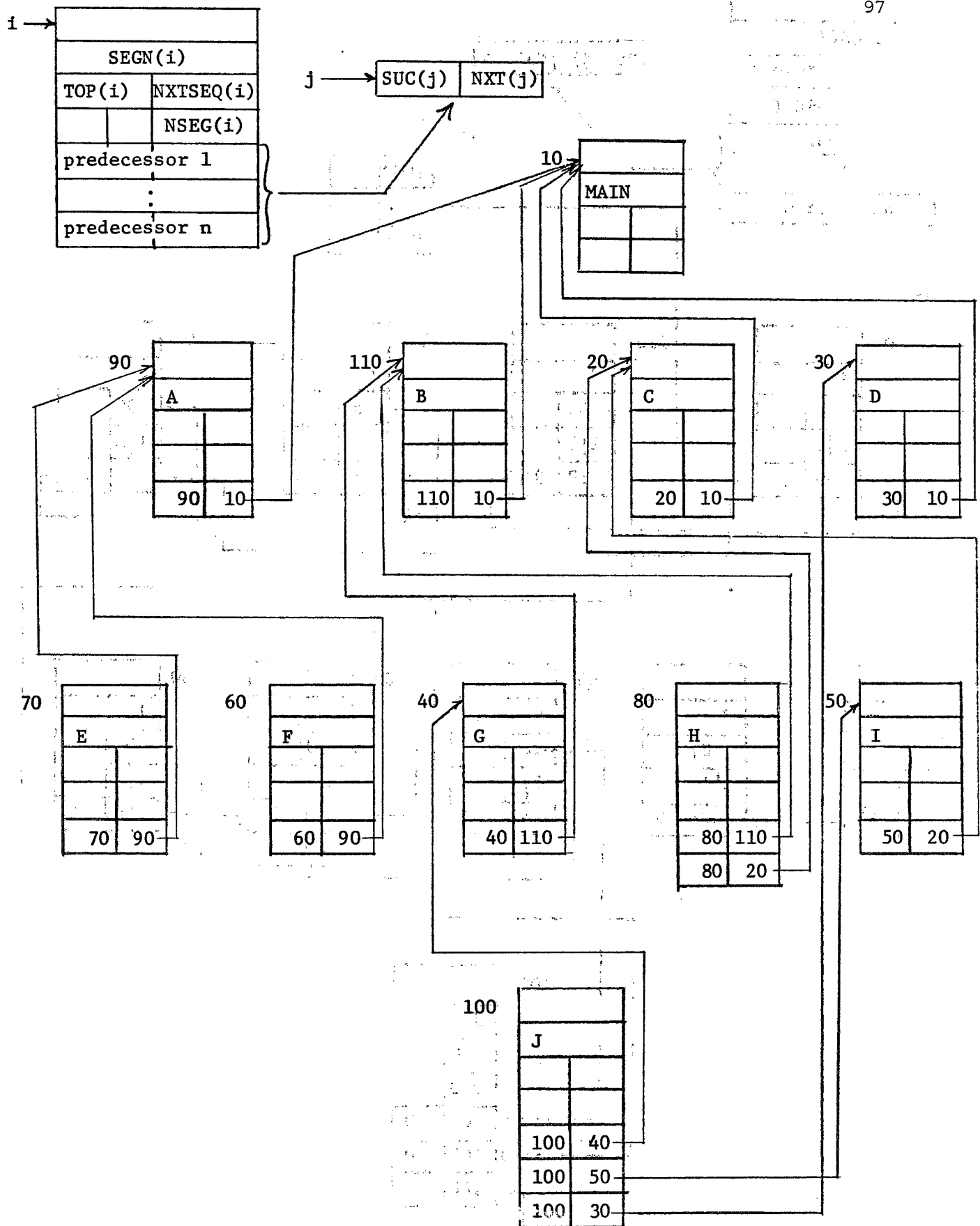


Figure 41

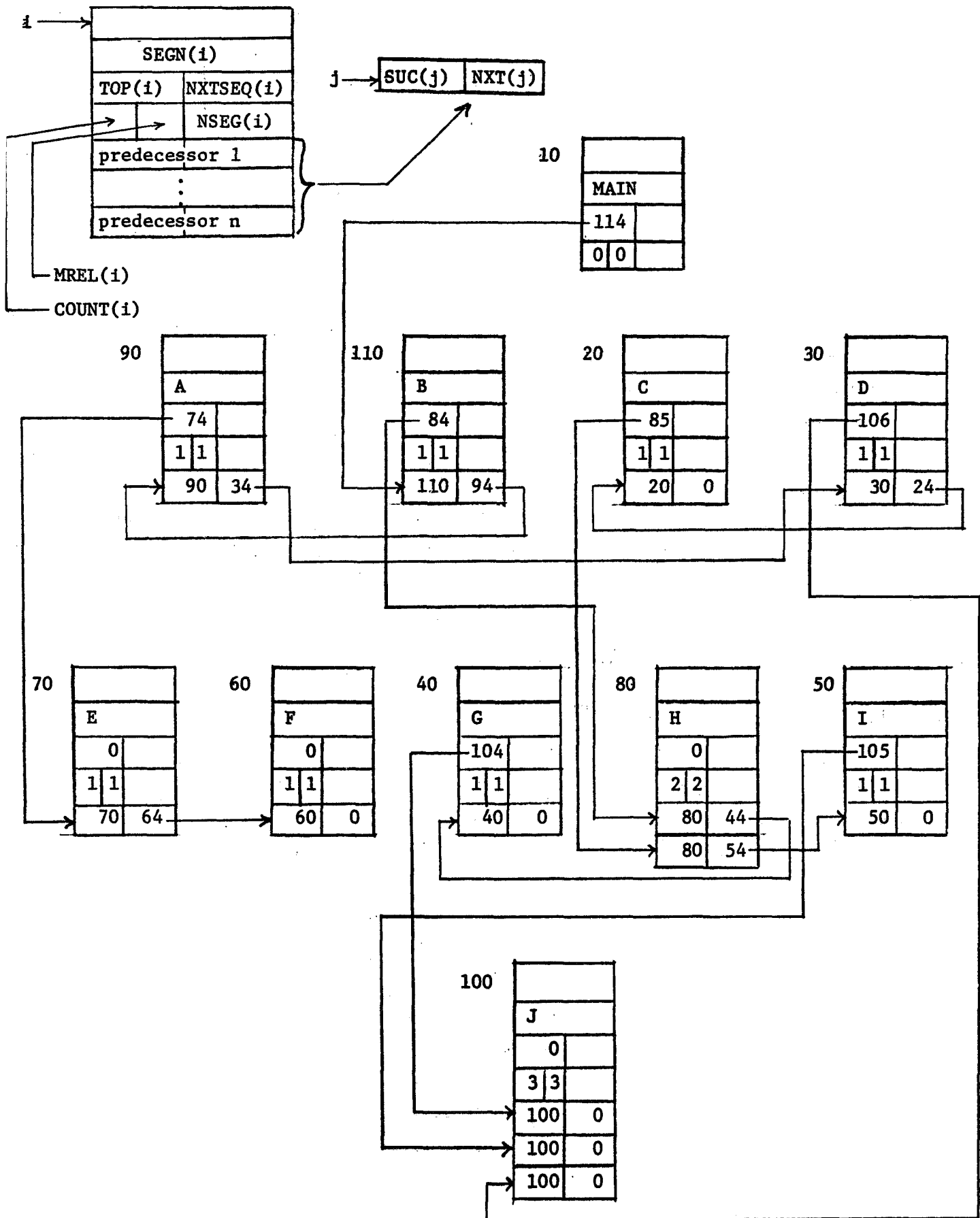


Figure 42

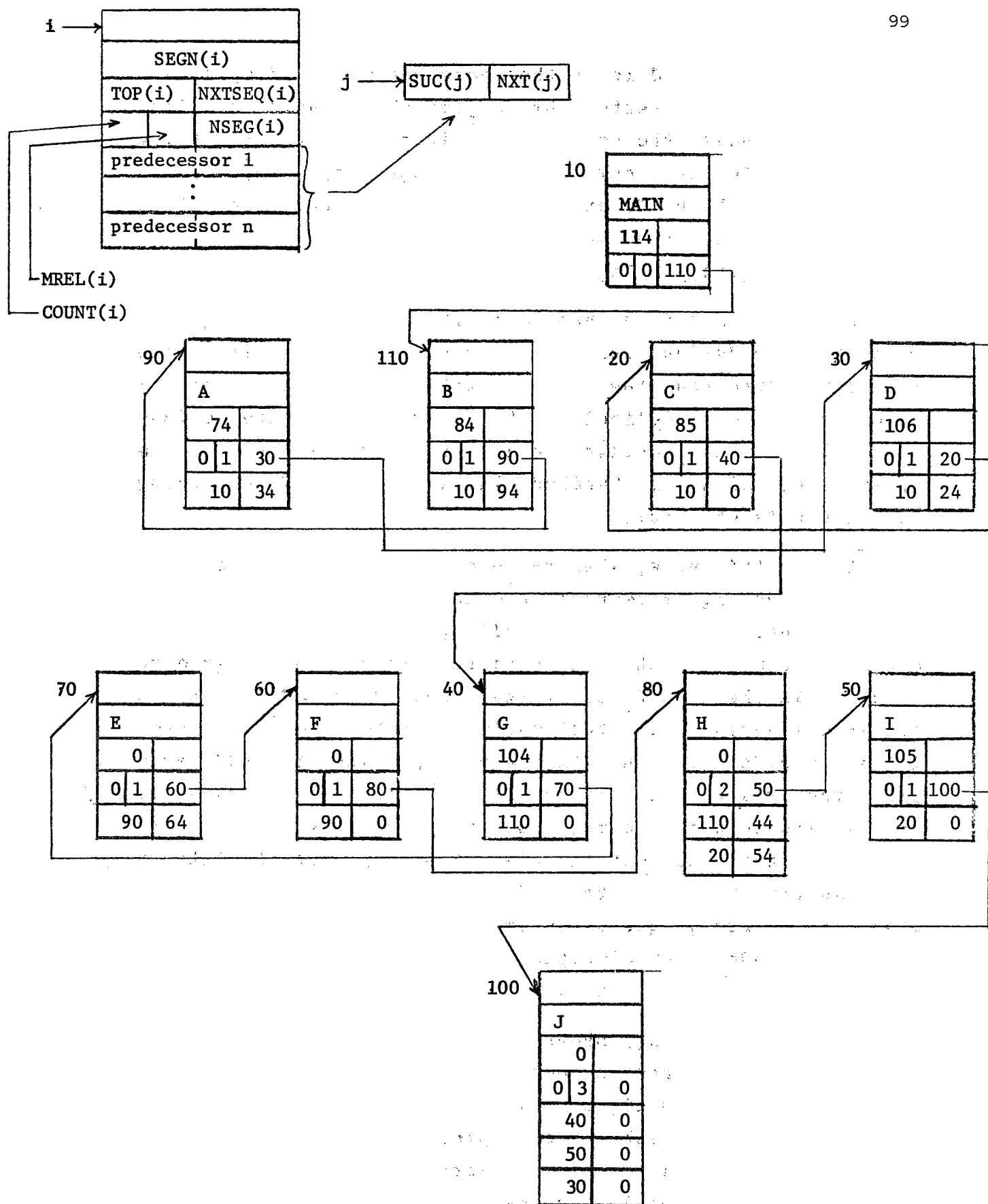


Figure 43



D, C. The segment J is on the successor chain of G, I, and E, but it is represented by separate one-word entries. Figure 43 shows the segment descriptors at the end of the sort. The NSFC field now links them together, MAIN, B, A, E, C, G, F, H, I, and J. It should be noted that this ordering does preserve the partial order.

#### Cross-referencing And Library Search

Once the user input has been exhausted and any required segment re-ordering has been accomplished, or after each library subprogram has been input, the loader cross-references the undefined symbol lists with the defined symbol table and resolves all unresolved symbols. The cross-referencing is handled in one of two ways depending upon the mode of input.

In the USER mode, when the end-of-file has been reached and the ordering of segment descriptors has been completed, the undefined symbol lists of all the user elements are cross-referenced with the defined symbol table (Figure 44(a)). In the LIBRARY mode, the undefined symbol list is cross-referenced with the defined symbol table as each library element is input (Figure 44(b)). The LIBRARY mode also requires that the first cell from the library queue be removed and returned to the list of available cells, and that the next relocatable element be input if the library queue is not empty. At the end of the user input mode and when the library queue is empty during the library input mode, the loader then attempts to resolve all unresolved symbols through the library.

Cross-referencing, shown in Figure 45, consists of replacing each symbolic address in the undefined symbol lists of element descriptor I with the address of the fourth word in defined symbol descriptor of the identical name. The fourth word will contain the absolute addresses assigned to the defined symbol. As some defined symbol entries may

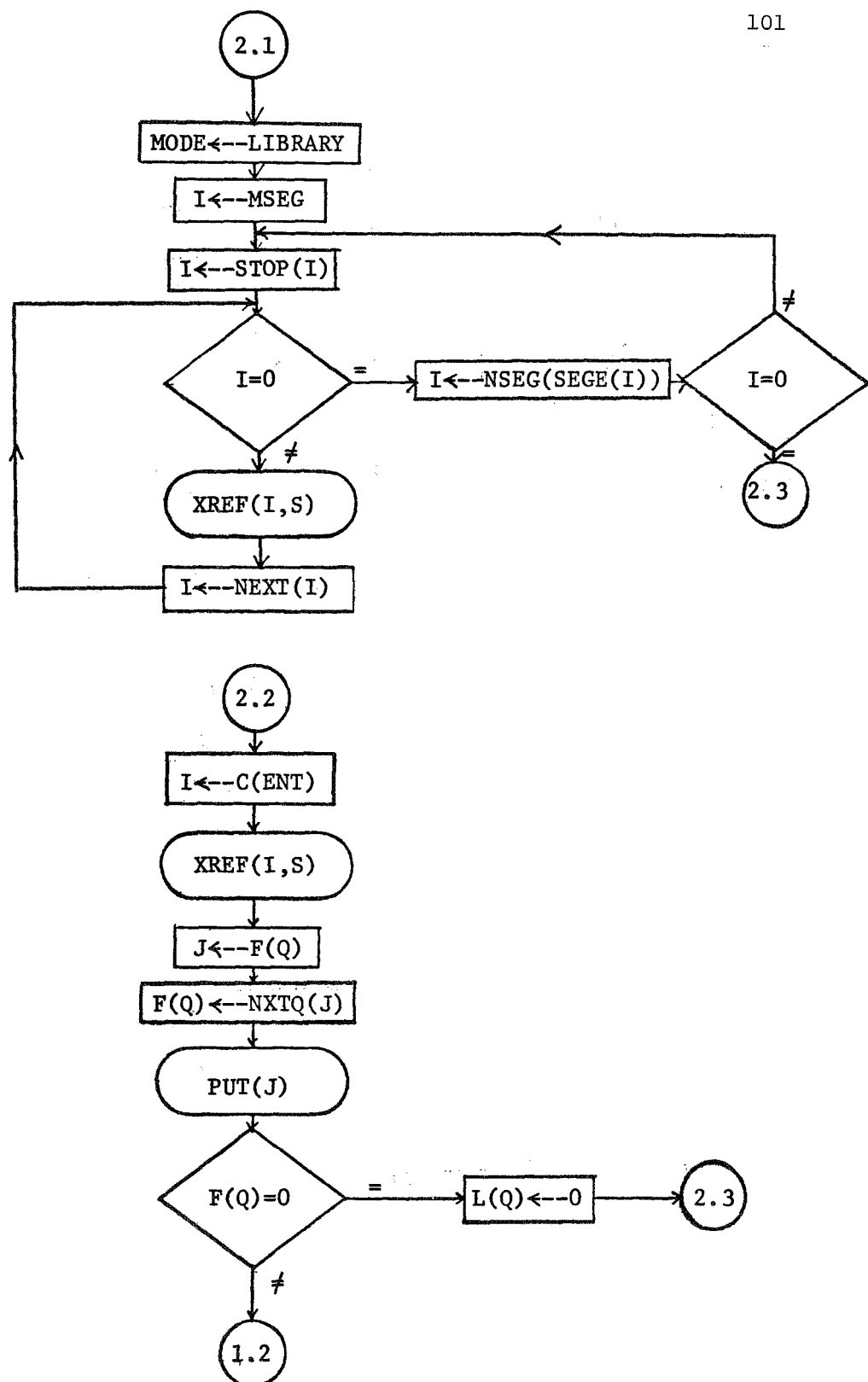


Figure 44



not be in the table, when a table search is unsuccessful a new entry is created in the table. In order to remember that this defined symbol is unresolved (i.e., does not yet correspond to an element), the defined symbol entry is added to a chain of unresolved symbols. In addition, when a reference is made from one element to an element in a different segment and the second element is a library element, the library element is moved to the main segment unless one of the two elements was already in the main segment. This assures that the loader will never introduce address references between segments that overlay each other. This also minimizes segment loading.

Whenever the unresolved symbol list is not empty and the library queue is empty, the unresolved symbols are resolved by searching the library index (Figure 46). The library index is input from mass storage location PLIB (initially IIB1), and the upper and lower bounds are stored in UB and LB, respectively. At this point the processing of the unresolved symbol chain begins. If a symbolic name of the defined symbol is within the range of the index ( $LB \leq DEFS(UL(I)) \leq UB$ ), then the index is searched for a matching entry. Whether the entry is found or not, the defined symbol is considered resolved. If it does not exist, then a diagnostic appears. If the entry is found, the library queue is searched to see if the element's mass storage location has been entered into the queue. If it is not in the queue, the mass storage location of the element and the segment to which the element should be attached are entered into the queue. If the mass storage location of the element is already in the queue, the segment is checked to make sure that if the element is to be referenced from different segments, that it will appear in the main segment. Once an element has been resolved, the defined symbol is unlinked from the unresolved symbol chain. When all of the defined symbols occurring in the range LB to UB have been resolved,

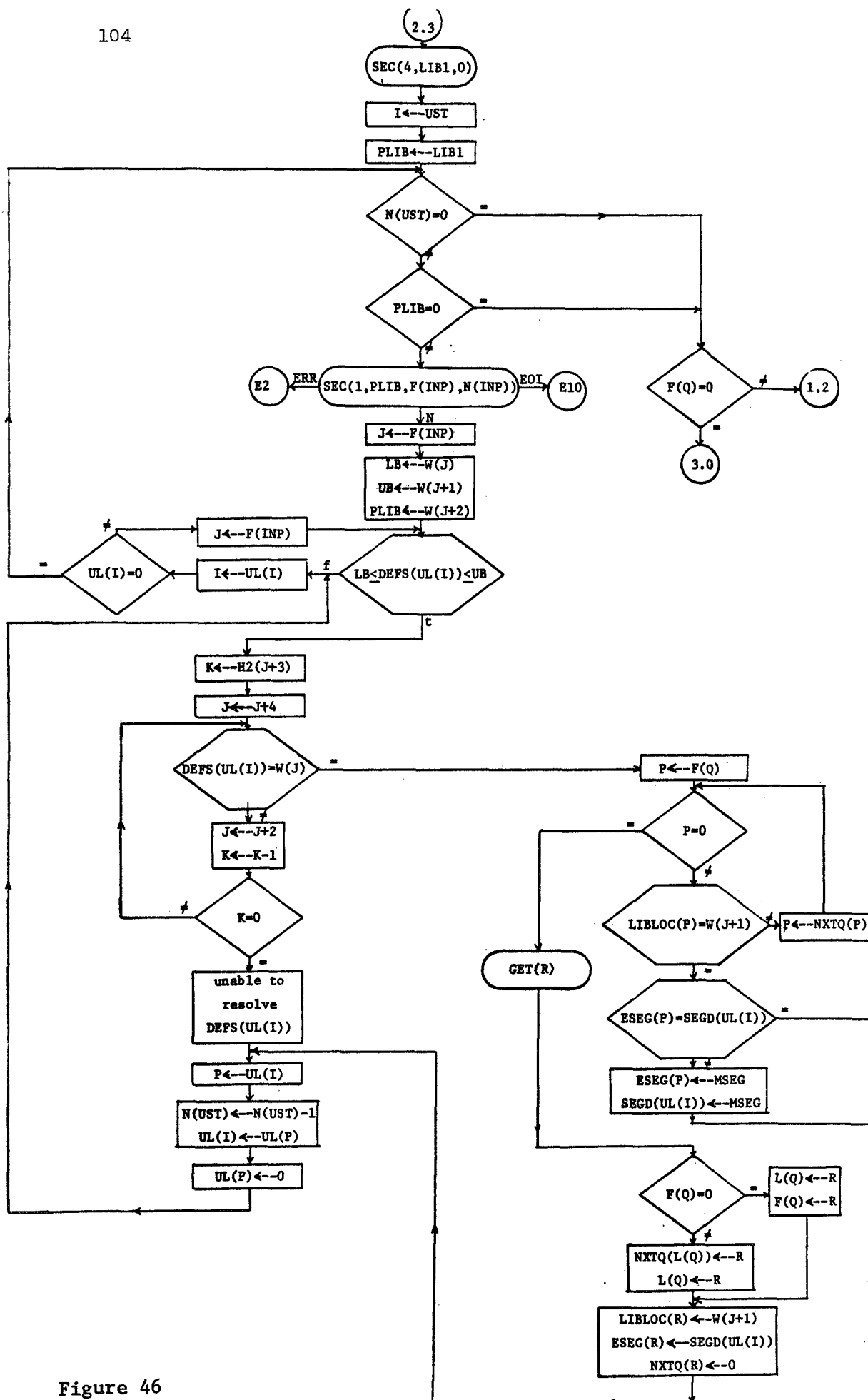


Figure 46

the next section of the library index is input. When all symbols have been resolved, the input of library elements begins. The process of inputting ends when the library queue and the unresolved symbol queues are both empty. This completes the first phase of the loader. All tables have been constructed, segments ordered, and links have been created between the defined symbol table and undefined symbol lists and between the common symbol lists and common area table.

### Example

Figure 47 shows the memory contents after cross-referencing the tables of the example in Figure 38. The undefined symbol list of subprogram MAIN (locations 1015<sub>8</sub> through 1017<sub>8</sub>) have now been loaded with the addresses of the fourth word in the defined symbol entries at locations 1024<sub>8</sub>, 1031<sub>8</sub>, and 1036<sub>8</sub>. These are the entries for SIN, CCS, SUBR, respectively. Because all of these entries are unresolved, the C field of the unresolved symbol table entry (UST) contains the address (1033<sub>8</sub>) of the first of a chain of three entries that are linked through the UI field. The defined symbol table consists of two chains. For the example, the hash function divides the alphabet in half (i.e., symbolic names beginning with letters A through M map to the first table location 776<sub>8</sub>, the rest map to the second table location 777<sub>8</sub>). The chain indicated at 776<sub>8</sub> consists only of the entry at 1026<sub>8</sub> (CCS). Note that the EI field of this entry is zero. The chain indicated at location 777<sub>8</sub> consists of two entries, at locations 1033<sub>8</sub> and 1021<sub>8</sub> (SUBR and SIN, respectively).

Figure 48 shows the memory content after resolving the unresolved symbols in the library index shown in Figure 12. The unresolved symbol table is now empty (the two-word cell at UST and the UI fields of the entries at 1021<sub>8</sub>, 1026<sub>8</sub>, and 1033<sub>8</sub> are zero). However, now the library queue, which is

776					1 0 2 6	
					1 0 3 3	
1000	\$	1 0 0 5		1 0 0 5		
		M	A	I	N	\$
						1
1005	M	1 0 0 0				0
		A	I	N	b b	
				1	1 0 2 0	
				3	1 0 1 5	
			1 2			0
	T	E	X	T	O	1
1015	S	I	N	0 0	1 0 2 4	
	C	O	S	0 0	1 0 3 1	
	S	U	B	0 0	1 0 3 6	
1020					7 7 7 6	
1021				0		0
	S	I	N	b b b		0
	4					
	4	1 0 0 0				
1026		1 0 2 1				0
	C	O	S	b b b		0
	4					
	4	1 0 0 0				
1033		1 0 2 6		1 0 2 1		
	S	U	B	R b b		0
	4					
	4	1 0 0 0				
7776	B	L	O	C	K b	
7777			1 0			0

TBL	7 7 7 6	1 0 0 0
	7 7 7 6	7 0 0 0
SEG	1 0 0 0	1 0 0 0
		1
ENT	1 0 0 5	
DST		7 7 6
		3
UST	1 0 3 3	
		3
ELT		
CAT	7 7 7 6	7 7 7 6
	7 7 7 6	
INP		1 0 0 0 0
	1 0 7 7 7	0 1 0 0 0
OUT		1 1 0 0 0
	1 1 7 7 7	0 1 0 0 0
AVAIL		1 1 0 0 0
	1 1 7 7 6	
Q		
MSEG		0
START	2	1 0 0 5

11000		1 1 0 0 2
		1 1 0 0 4
		1 1 0 0 6
		1 1 0 1 0
		1 1 0 1 2
		1 1 7 7 2
		1 1 7 7 4
		1 1 7 7 6
11777		0

Figure 47

776					1 0 2 6
					1 0 3 3
1000		1 0 0 5		1 0 0 5	
	\$	M	A	I	N \$
					1
1005		1 0 0 0			0
	M	A	I	N	b b
			1		1 0 2 0
			3		1 0 1 5
			1 2		0
	T	E	X	T	0 1
1015	S	I	N		1 0 2 4
	C	O	S		1 0 3 1
	S	U	B		1 0 3 6
1020					7 7 7 6
1021			0		0
	S	I	N	b b b	0
	1				
		1 0 0 0			
1026			0		0
	C	O	S	b b b	0
	1				
		1 0 0 0			
1033			0		1 0 2 1
	S	U	B	R	b b
	1				0
		1 0 0 0			
7776	B	L	O	C	K b
7777			1 0		0

TBL	7 7 7 6	1 0 0 0
	7 7 7 6	7 0 0 0
SEG	1 0 0 0	1 0 0 0
		1
ENT	1 0 0 5	
		1
DST		7 7 6
		3
UST	0	
		0
ELT		
CAT	7 7 7 6	7 7 7 6
	7 7 7 6	
INP		1 0 0 0 0
	1 0 7 7 7	1 0 0 0
OUT		1 1 0 0 0
	1 1 7 7 7	1 0 0 0
AVAIL		1 1 0 0 4
	1 1 7 7 6	
Q		1 1 0 0 0
	1 1 0 0 2	
MSEG		0
START	2	1 0 0 5

11000	L	I	B	L	C	1
		1	0	0	0	2
	L	I	B	L	C	2
		1	0	0	0	0
					1	1
					0	0
					6	
					1	1
					0	1
					0	
					1	1
					0	1
				2		
				1	1	
				7	7	
				2		
				1	1	
				7	7	
				4		
				1	1	
				7	7	
				6		
11777					0	

Figure 48



addressed by the F and I fields at Q, contains two entries, IIBIC1 at address 11000<sub>8</sub>, and IIBIC2 at address 11002<sub>8</sub>.

Figure 49 shows the memory content after completing all input, user and library. There are now three element descriptors, at locations 1005<sub>8</sub>, 1040<sub>8</sub>, and 1066<sub>8</sub>, representing subprograms MAIN, SUBRTN, and SINCCS, respectively. The subprograms SUBRTN and SINCCS are shown as relocatable elements in Figures 8 and 9. Another defined symbol table entry has been added at location 1053<sub>8</sub> for the symbolic address CNSTNT in subprogram SUBRTN. All of the defined symbol table entries have been completed, with their relative addresses appearing in the RELAD field. Two of the defined symbols, SIN and CCS, are relative to the subprogram SINCCS. This is indicated by the fact that the DSL field of the SINCCS element descriptor points to location 1021<sub>8</sub>, the SIN entry, and the ELTL field of this entry points to 1026<sub>8</sub>, the CCS entry. The other two defined symbols, SUBR and CNSTNT, are relative to subprogram SUBRTN.

TBL	7776 1000	776	1053	1050	S I N 1024	109
	7776 7000		1033		C O S 1031	
SEG	1000 1000	1000	1005 1066		C N S 1056	
	1		\$ M A I N \$	1053	0 1026	
ENT					C N S T N T	
	3				000000077777	
DST	776				0 0	
	4	1005	1000 1040		1000 0	
UST	0		M A I N b b	1060	7776	
	0		1 1020	1061	1063	
ELT			3 1015			
			0 0	1063	12 231 0	
CAT	7776 7776		0 0		60 012 2	
	7776		12 0		31 160 1	
INP	10000		T E X T 0 1	1066	1000 0	
	10777	1015	S I N 1024		S I N C O S	
OUT	11000		C O S 1031		0 0	
	11776		S U B 1036		0 0	
AVAIL	11004	1020	7776		0 0	
	11002	1021	0 0		2 1021	
Q	0		S I N b b b		50 0	
	11002		0		T E X T 0 3	
MSEG	1000		1 5 0			
START	2 1005		1000 1026	7776	B L O C K b	
BASE	1000	1026	0 0		12 0	
			C O S b b b	11000		
			1 0 0		11002	
			1000 0		0	
		1033	0 1021		11006	
			S U B R b b		11010	
			0			
			1 10 0			
			1000 1053			
		1040	0 1021			
			S U B R T N			
			1 1060			
			3 1050	11774	11774	
			2 1061		11776	
			2 1033			
			200 0			
			T E X T 0 2	11777	11000	

Figure 49

### 4.2.3 Global Allocation

After all elements have been input, the loader proceeds to phase two, global allocation. In this phase, the absolute addresses of all common areas, subprograms, defined symbols, and complex addresses are computed. By computing absolute addresses, memory is allocated to the common areas and subprograms. Figure 50 shows the basic flow of this phase. After the common areas are allocated space, each segment is allocated space in the order that resulted from the topological sort. A segment is allocated by assigning each subprogram within it an absolute address. As each element is allocated, the defined symbols and common areas that occur within it are assigned absolute addresses. Each segment is assigned the memory space following the largest of its predecessor segments. Once all segments are allocated, the complex calculation tables are processed to provide the complex addresses with values. The global allocation is terminated by assigning the starting address. It should be noted that global allocation requires only the data that exists in the tables.

#### Allocate Undefined Common Areas

Figure 51 shows the allocation of undefined common areas between 3.0 and 3.0B in the flow chart. The variable ABSAD is used throughout the phase as the next available memory location. This is initialized with the value of BASE, the lowest memory address available to the user. The undefined common area must be allocated in the resident part of the program because of the overlay structure. The absolute location of each common area is placed in that entry's CAESLC field, and the value of ABSAD is increased by MSIZ. A test is made each time ABSAD is increased to see if the highest available address, TSPACE, has been exceeded.

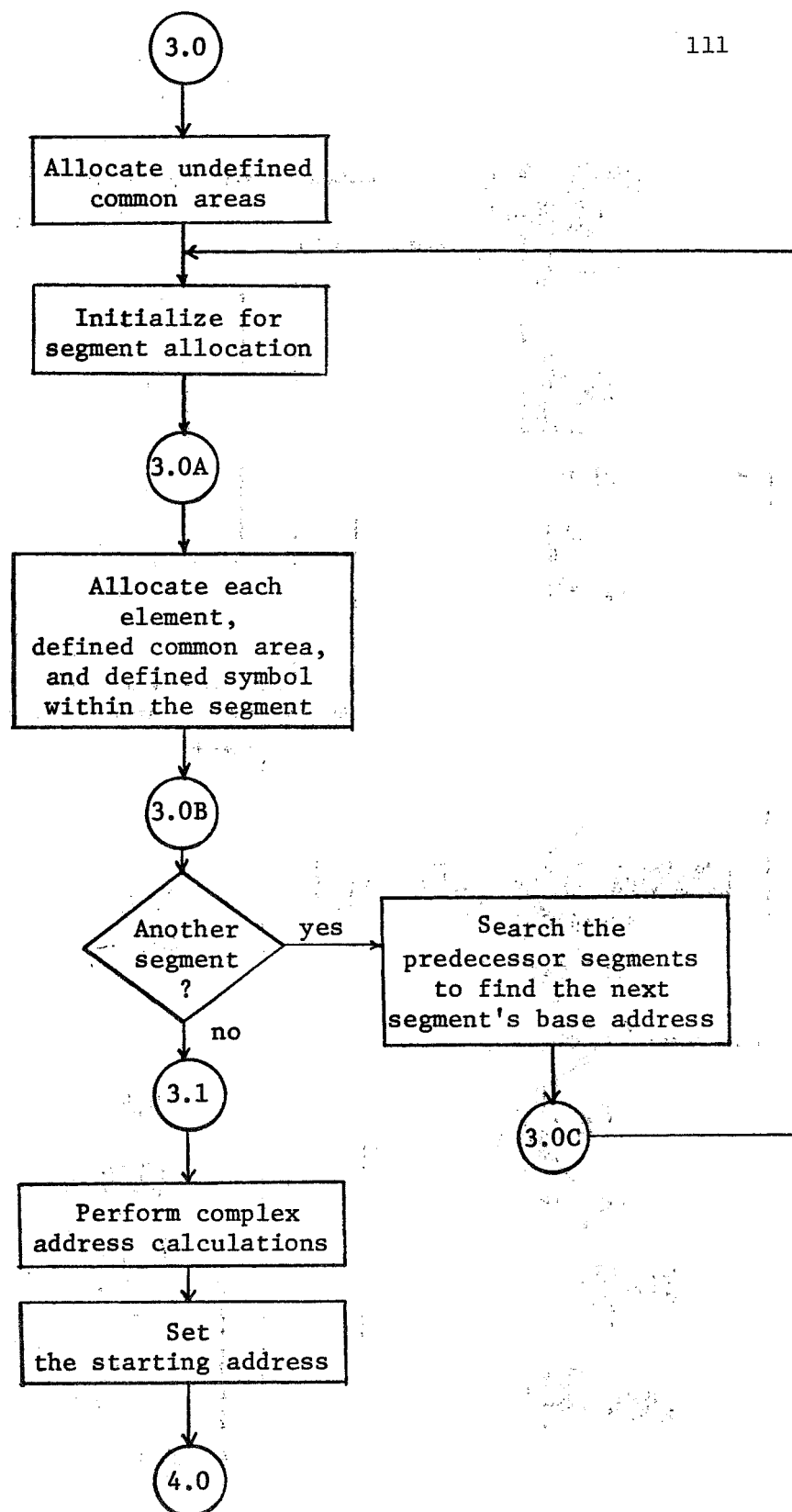


Figure 50 Flow chart Showing Global Allocation

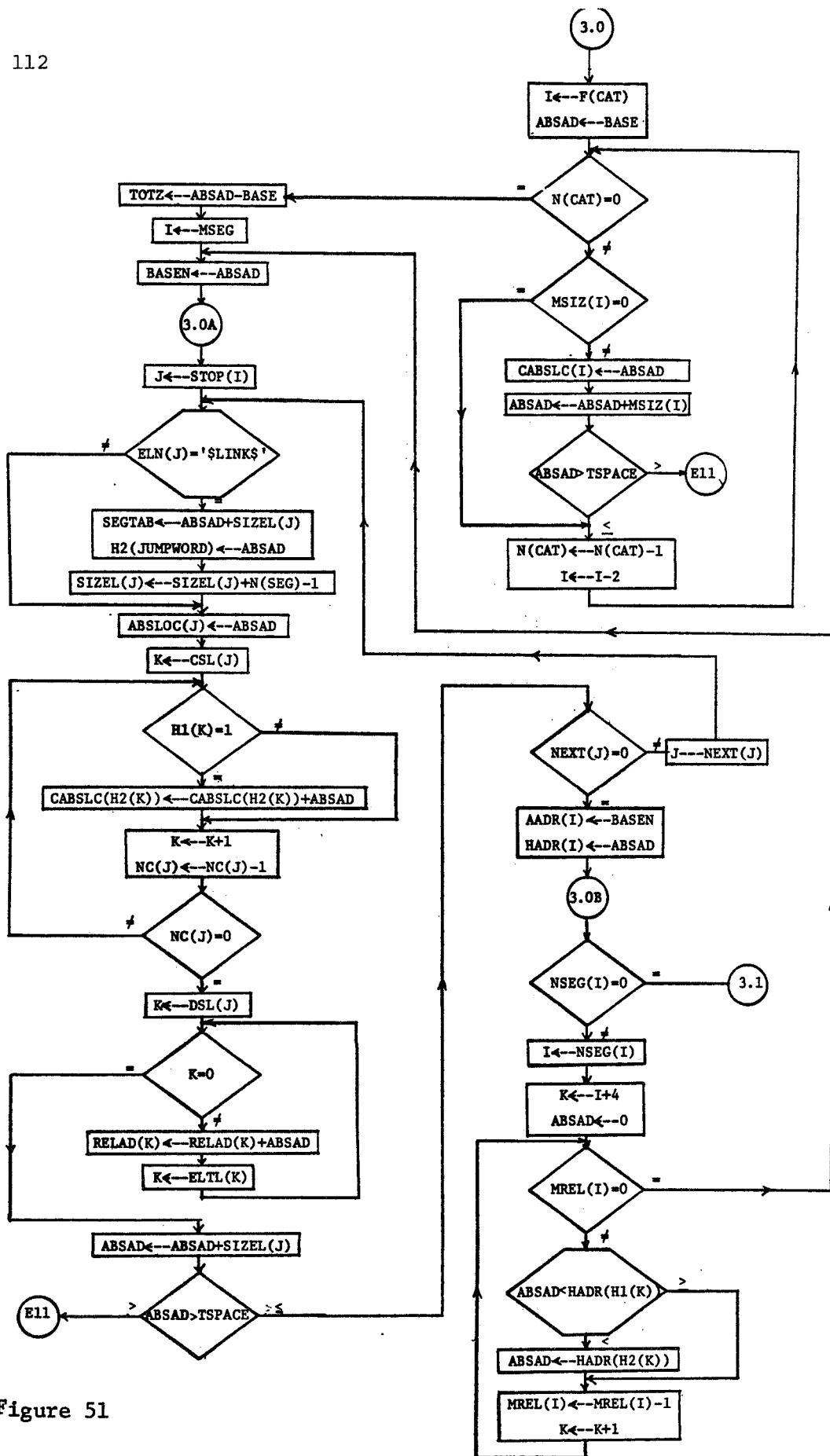


Figure 51

When all common areas have been allocated ( $N(CAT)=0$ ), then the total number of words allocated to undefined common is placed in TOTZ. The address of the first segment to be allocated, MSEG, is placed in I.

#### Allocate The Subprograms

The allocation of one segment is shown between 3.0A and 3.0E. The flow chart first tests for the existence of the segment loading subprogram, \$LINK\$. The presence of this will be ignored for now. Each element is assigned an absolute address in its ABSLOC field. The list of common areas for each element is scanned to see if any are defined within the element ( $H1(K)=1$ ). Those are assigned an absolute address by adding the relative address stored in CABSIC to the absolute address ABSAD. Next, the list of defined symbols is processed. For each entry the relative address in the RELAD field is added to the absolute address assigned to the element, ABSAD. When each element has been allocated, the value of ABSAD is increased by the value of the SIZE1 field of that element descriptor, the test for memory overflow is made, and the next element descriptor is located. Once all elements in a segment have been allocated ( $NEXT(J)=0$ ), the address assigned to the segment is stored in the AADR field of the segment descriptor, and the next available address is stored in the HADR field. This defines the extent of the segment.

If the user program is segmented, then memory overlay is indicated, and the library routine that performs the segment overlay, named \$LINK\$ herein, has been added to the main segment's chain of elements. This subprogram is allocated memory in a different manner, and therefore, the algorithm scans for this subprogram specifically. When it discovers it, it stores the absolute address immediately following it in variable SEGTAB and increases the element size indicated for it by the number of segments minus 1.

This is done so that a table of segment locations may be loaded into the memory immediately following the subprogram \$LINK\$.

Before the next segment can be processed, the HADR fields of all the predecessors must be compared to find the largest value. The last part of the flow chart describes the tests necessary to accomplish this.

#### Compute The Complex Addresses

Once all segments have been allocated, the complex addresses are calculated (Figure 52). This requires that each element descriptor be scanned to see if it contains a complex address list. As mentioned, the first word of the complex address list contains the address of the table of complex calculations for that list. The table is a sequence of half-word instructions that define the calculations to be performed for each complex address. In order to unpack the instructions, the algorithm shown in Figure 52 uses a switch SW to indicate which half-word is being executed. Once the instruction has been unpacked into the operator field OPC, the table indicator field TI, and the table index field INDEX, then the address or value of the symbolic address at location index in table TI is retrieved by the FIND function.

Figure 53 shows the flow chart for the FIND function. If TI is 1, then the address of the defined symbol pointed to at location INDEX of the undefined symbol list is stored in VAL. If TI is 2, then the contents of the VALUE field of the defined symbol is stored in VAL. If TI is 3, then the address of the common area pointed to at location INDEX of the common symbol list is stored in VAL. If TI is 4, then the complex address located at location INDEX in the complex address list is stored in VAL. If TI is 5, then the value of the index is loaded into VAL. If TI is 0, 6, or 7, nothing happens.

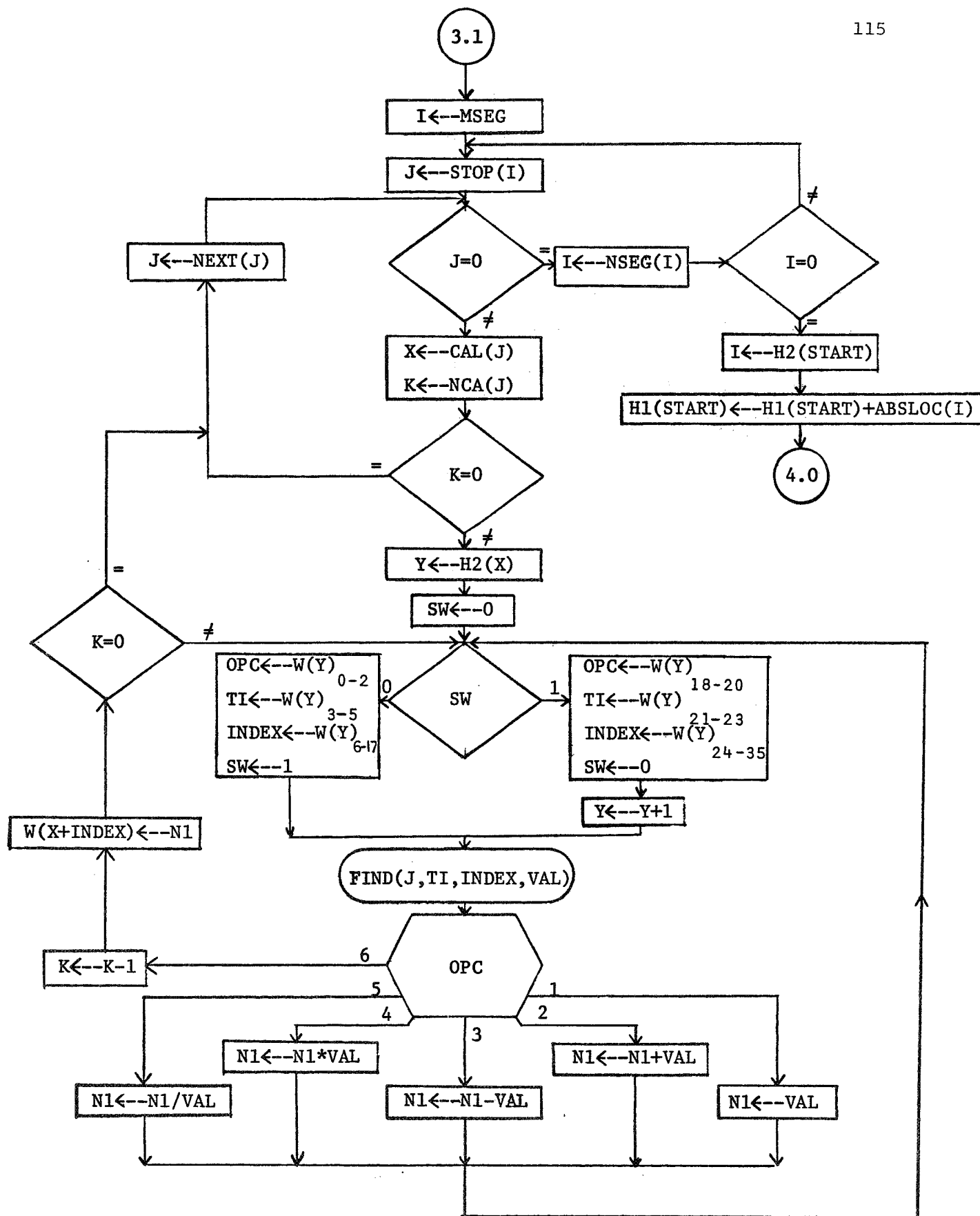


Figure 52



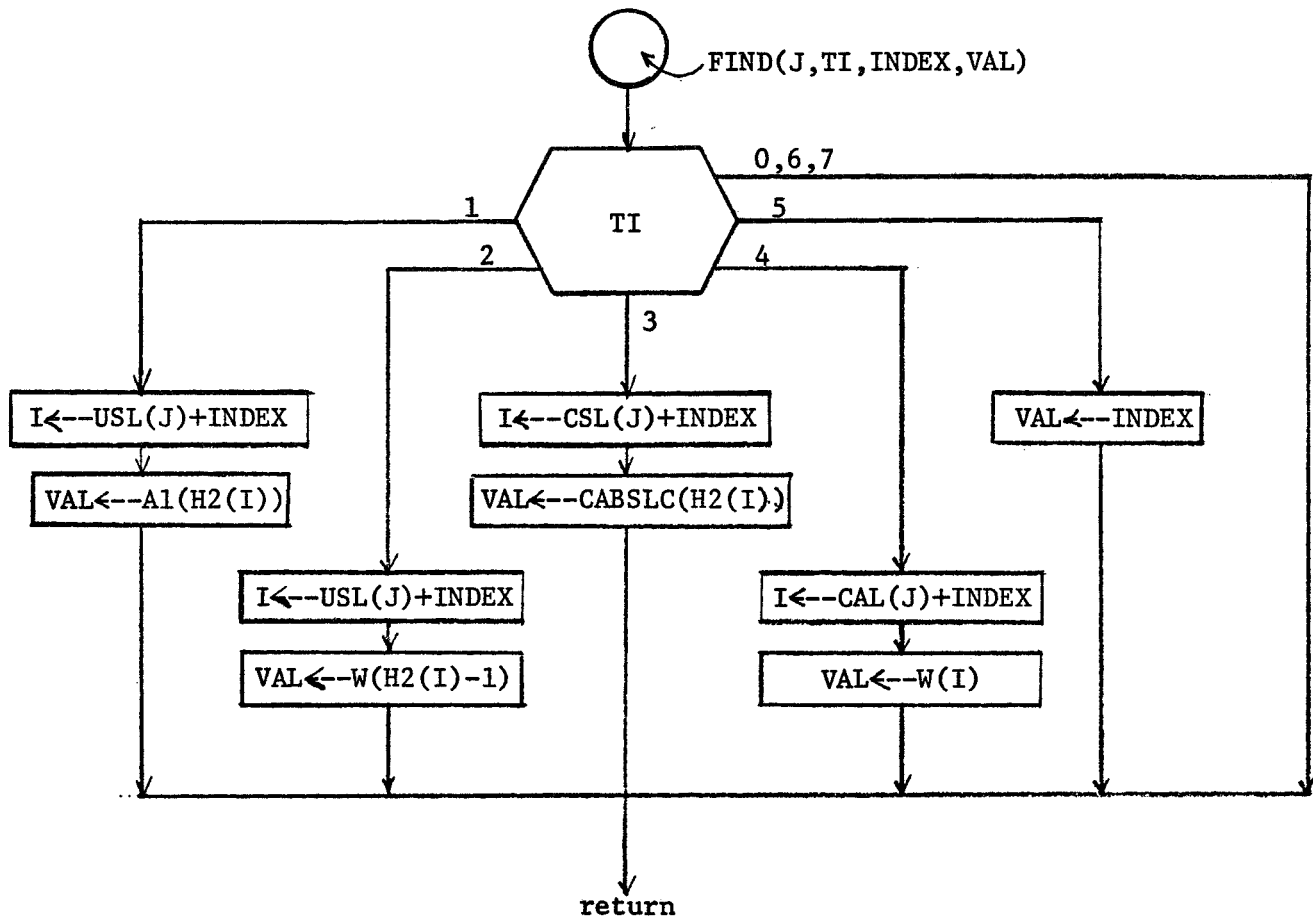


Figure 53

Once the variable VAI has been set by the FIND function the operation specified by the operator field CPC is carried out:

- (a) if OPC is equal to 1, then VAI is loaded into N1;
- (b) if CPC is equal to 2, then VAL is added to N1;
- (c) IF CEC is equal to 3, then VAL is subtracted from N1;
- (d) if OPC is equal to 4, then N1 is multiplied by VAL;
- (e) if CPC is equal to 5, then N1 is divided by VAL;
- (f) if CPC is equal to 6, then N1 is stored in the complex address list at location INDEX, and the calculation is complete.

The complex address list of each element is processed in this manner.

Once all complex addresses have been calculated, the absolute address of the starting address is computed and stored in the first half of variable START. At this point global allocation is complete, and the processing of relocatable code begins.

#### Example

Figure 54 shows the contents of memory after global allocation for the example. The common area BLOCK (descriptor at location 7776<sub>8</sub>) has been assigned absolute address 1000<sub>8</sub>. The subprograms MAIN, SUBRTN, and SINCCS (element descriptors at 1005<sub>8</sub>, 1040<sub>8</sub>, and 1066<sub>8</sub> respectively) have been assigned absolute addresses 1012<sub>8</sub>, 1024<sub>8</sub>, and 1224<sub>8</sub> because the common area required 12<sub>8</sub> locations, subprogram MAIN required 12<sub>8</sub> locations, and subprogram SUBRTN required 200<sub>8</sub> locations. Defined symbols SIN and COS representing symbolic addresses at relative locations 5 and 0 within subprogram SINCCS, have been assigned absolute addresses 1231<sub>8</sub> and 1224<sub>8</sub>, respectively. Defined symbols CNSTNT and SUBR, representing symbolic

TBL	7 7 7 6	1 0 0 0	776		1 0 5 3	1050	S I N	1 0 2 4
	7 7 7 6	7 0 0 0			1 0 3 3		C O S	1 0 3 1
SEG	1 0 0 0	1 0 0 0	1000	1 0 0 5	1 0 6 6		C N S	1 0 5 6
		1		\$ M A I N \$		1053	0	1 0 2 6
ENT	1 0 6 6			1 0 0 2	1 2 7 4		C N S T N T	
		3			1		0 0 0 0 0 0 0 7 7 7 7 7	
DST		7 7 6					1 0 2 4	0
		4	1005	1 0 0 0	1 0 4 0		1 0 0 0	0
UST	0			M A I N b b		1060		7 7 7 6
		0		1	1 0 2 0	1061		7 6 5 4 6
ELT				3	1 0 1 5			7 6 5 5 3
				0	0	1063	1 2	2 3 1 0
CAT	7 7 7 6	7 7 7 6		0	0		6 0	0 1 2 2
	7 7 7 6			1 2	1 0 1 2		3 1	1 6 0 1
INP		1 0 0 0 0		T E X T 0 1		1066	1 0 0 0	0
	1 0 7 7 7		1015	S I N	1 0 2 4		S I N C O S	
OUT		1 1 0 0 0		C O S	1 0 3 1		0	0
	1 1 7 7 6			S U B	1 0 3 6		0	0
AVAIL		1 1 0 0 4	1020		7 7 7 6		0	0
	1 1 0 0 2		1021	0	0		2	1 0 2 1
Q		0		S I N b b b			5 0	1 2 2 4
	1 1 0 0 2				0		T E X T 0 3	
MSEG		1 0 0 0		1 1 2 3 1	Q			
START	1 0 1 4	1 0 0 5		1 0 0 0	1 0 2 6			
BASE		1 0 0 0	1026	0	0	11000	5 5 5 5 5 5 0 0 1 0 1 2	
TOTZ		1 2		C O S b b b			2 0 0 4 0 0 0 0 0 0 0 0	
BASEN		1 0 1 2			0		2 0 0 4 0 0 0 0 0 0 0 0	
TSPACE		7 7 7 7 7		1 1 2 2 4	0		0 0 7 4 0 0 4 0 1 2 2 4	
MIND		5 5 5 5 5 5		1 0 0 0	0		0 0 0 0 0 0 0 0 1 0 1 2	
			1033	0	1 0 2 1		0 6 0 1 0 0 0 0 1 0 1 2	
				S U B R b b			0 0 7 4 0 0 4 0 1 2 3 1	
					0		0 0 0 0 0 0 0 0 1 0 1 3	
				1 1 0 3 4	0		0 3 0 0 0 0 0 0 1 0 1 2	
				1 0 0 0	1 0 5 3		0 7 7 4 0 0 4 0 1 0 1 7	
			1040	1 0 0 0	1 0 6 6		2 0 1 0 0 5 3 0 1 0 3 4	
				S U B R T N				
				1	1 0 6 0			
				3	1 0 5 0			
				2	1 0 6 1			
				2	1 0 3 3			
				2 0 0	1 0 2 4			
				T E X T 0 2				

Figure 54

addresses at relative locations 0 and  $10_8$  within subprogram SUBFTN, have been assigned absolute addresses  $1024_8$  and  $1034_8$ , respectively. The complex addresses at locations  $1061_8$  and  $1062_8$  have been calculated and contain values  $76546_8$  and  $76553_8$ .

#### 4.2.4 Rellocatable Translation And Output

The third and last phase of the loader algorithm is to translate all the relocatable code to executable code, using the absolute addresses assigned in phase two, and to output the absolute element. Figure 55 shows the flow chart for this phase. In the case of memory overlay, seven tasks must be performed:

- (a) setup and output the header record;
- (b) translate the relocatable code for each element in the overlay segments and output the executable code;
- (c) translate the relocatable code for each element in the main segment and output the executable code;
- (d) assign a linkage table address for each global address that is referenced from another segment;
- (e) setup and output the segment table;
- (f) combine and output the linkage table;
- (g) perform all necessary print-out.

If there is no overlay, then only steps (a), (c), and (g) are performed.

The output of the header record is shown as a detailed flow chart in Figure 56. The header record consists of four words, containing the name of the absolute element (ANAME), the location and number of undefined common area words (BASE and TOTZ), the starting address (H1(START)), and the main

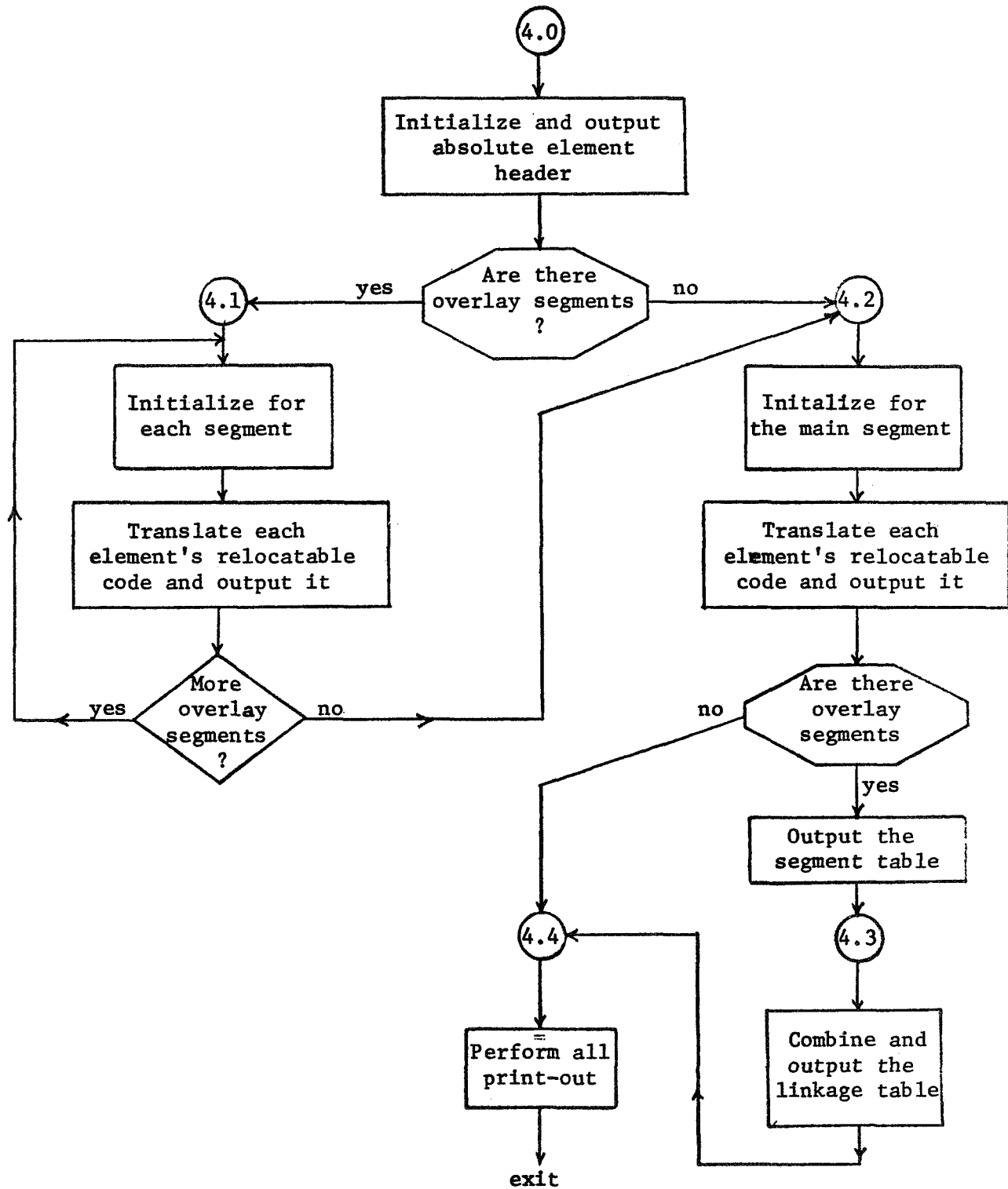


Figure 55 Flow chart Showing Relocatable Translation and Output

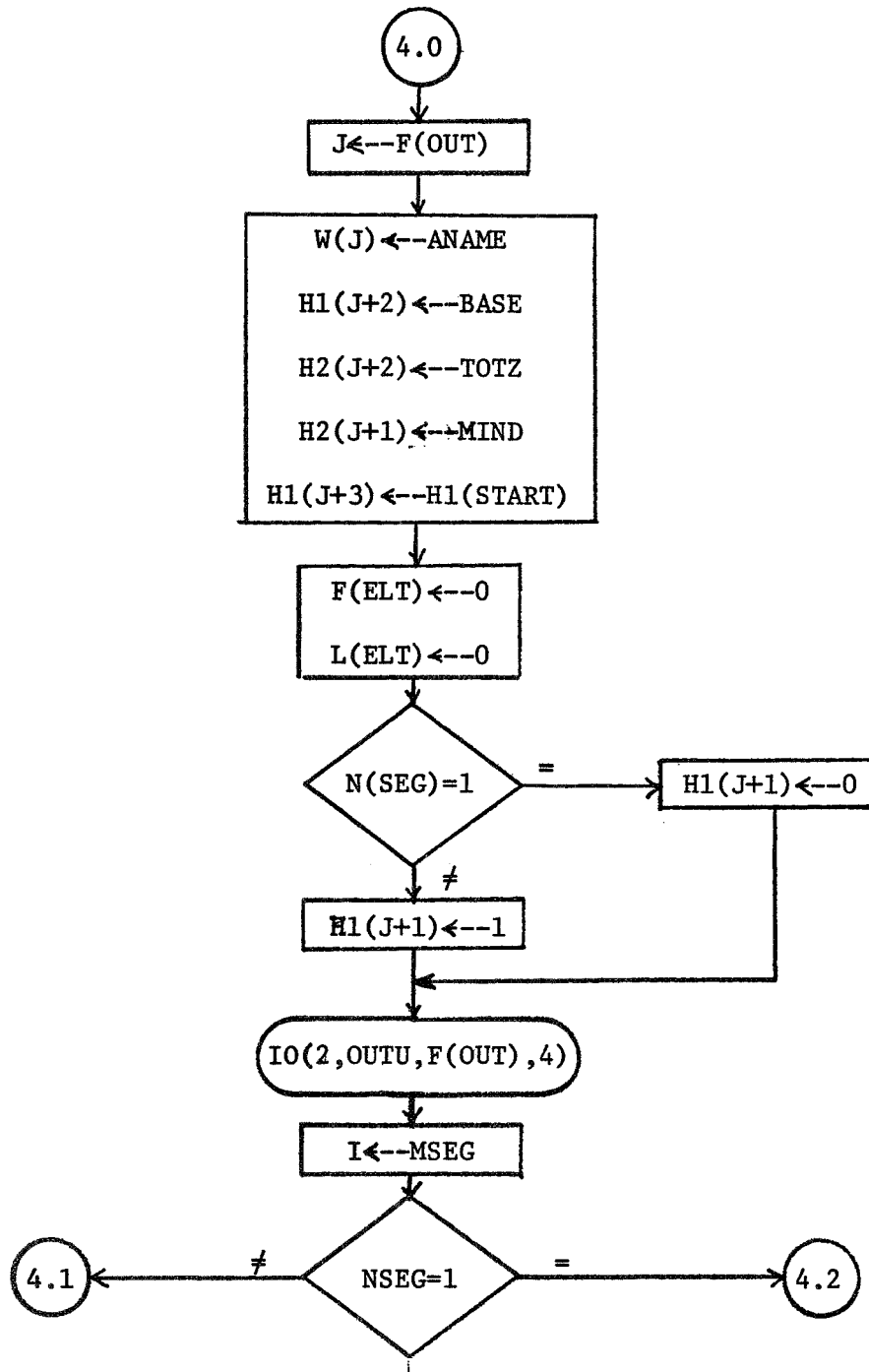


Figure 56

segment indicator flag (MIND). In addition, the first half of the second word is set to 1 if overlay segments follow. The record is output to I/C unit CUTU.

The processing of overlay segments is shown as a flow chart in Figure 57. Each segment is initialized by setting the segment indicator INDIC to zero, setting the ELC field of the segment descriptor to zero, storing the current number of overlay segment records in the CUTLOC field of the segment descriptor, and loading J with the address of the first element descriptor. For each element, processing is initialized by loading the input buffer (first location is F(INP)) with the first block of relocatable code from element's temporary mass storage location MASLOC(J). The relocatable code for the element is translated and output by the function TRANS which is explained in detail below. Once the translation of the relocatable code is complete, the mass storage location MASLOC(J) is released to the secretary, the next element descriptor is located, and the element processed. When all the elements for the segment are processed, the next segment descriptor is located and the next segment is processed. When all segments have been processed, the variable I is loaded with the address of the segment descriptor for the main segment.

The flow chart in Figure 58 describes the processing of the main segment and, in the case of overlay, the creation and output of the segment table. The former is essentially the same as for overlay segments except that the segment indicator, INDIC, is loaded with the main segment indicator flag, MIND. The translation of the relocatable code by the function TRANS is discussed later.

The creation and output of the segment table occurs only if more than one segment was input; otherwise, the algorithm is complete except for writing an end-of-file mark on the output unit CUTU. In the case of overlay, the

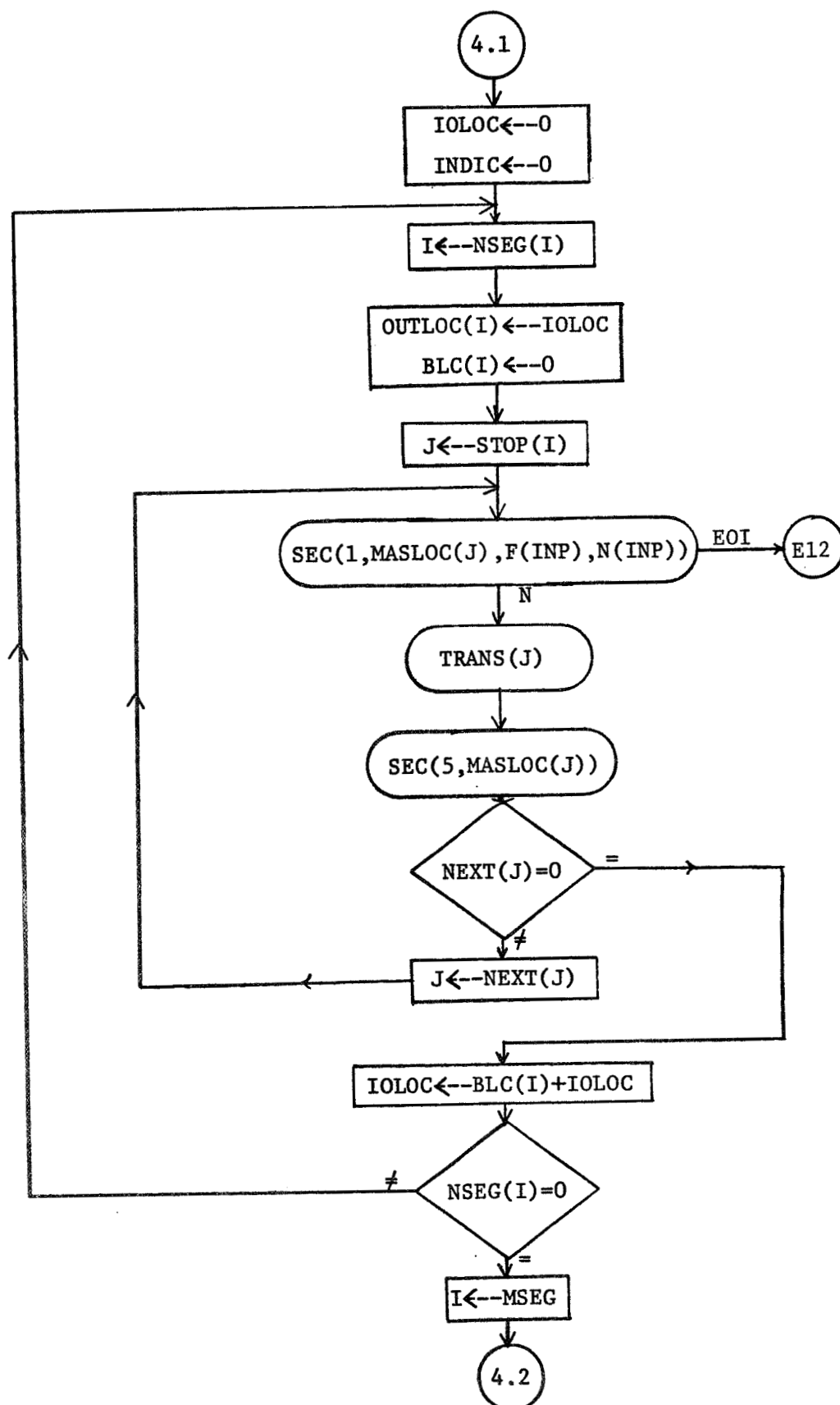


Figure 57



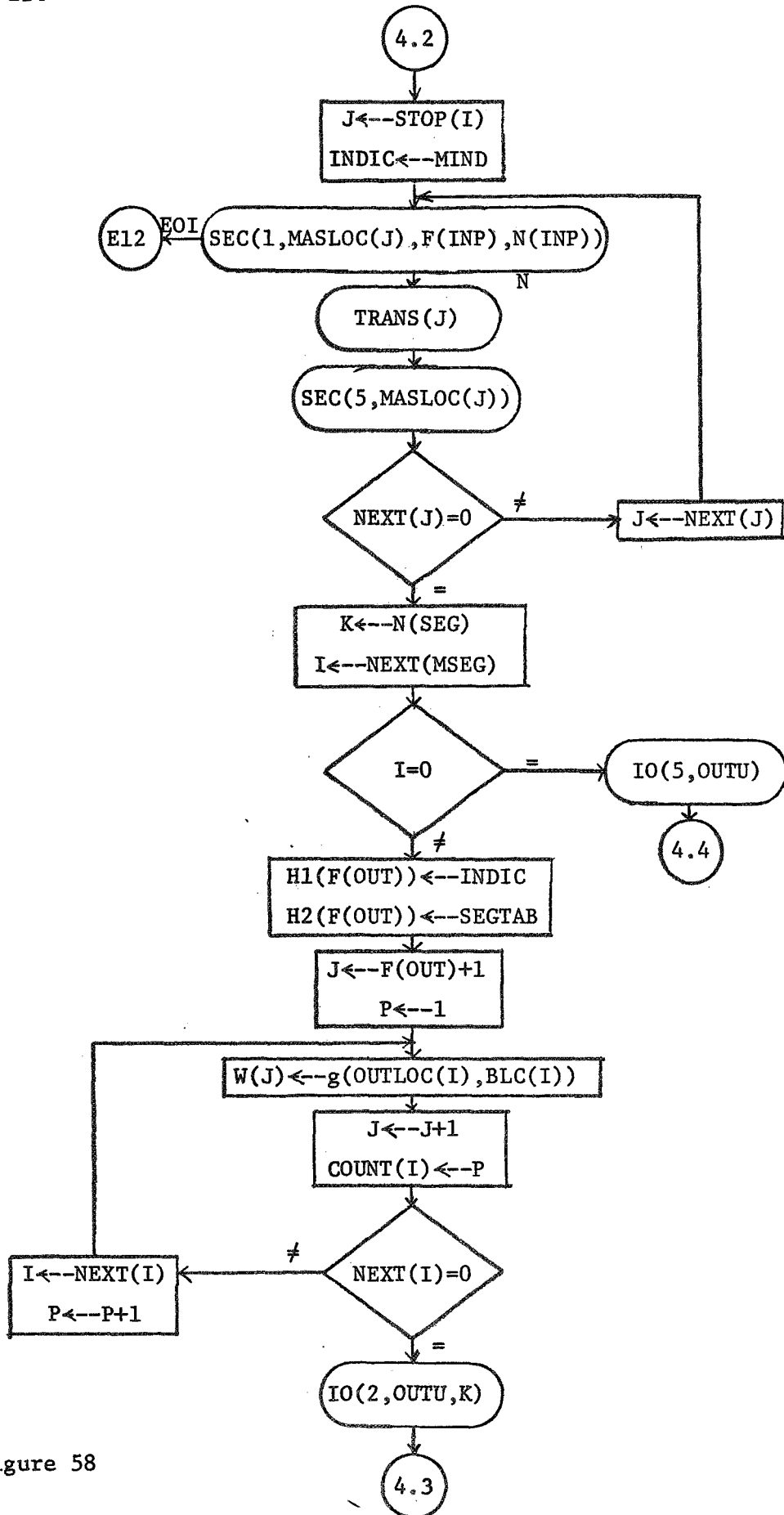


Figure 58

overlay segment loading subprogram \$LINK\$ requires a segment table defining the location and size of each overlay segment. For this algorithm linear files are assumed, and therefore the record location CUTLCC(I) and the number of records BIC(I) in the segment I will satisfy the requirements. The table must be placed immediately following the \$LINK\$ subprogram at the address stored in SEGTAB. The output is formatted as a scatter load record with the variables INDIC and SEGTAB in the first word and the segment table following. The segment table is made up of one-word entries. The format of the \$LINK\$ is not specified within this report. Instead, an undefined mapping  $g$  (CUTLCC(I), ELC(I)) is used to construct the entries, with the understanding that the definition of \$LINK\$ includes the definition of  $g$ .

The creation and output of the linkage table is shown in Figure 59. A part of the TRANS function has created a chain of global addresses that are referenced from other segments, and assigned each a unique address in the linkage table. This chain exists within the defined symbol table, but can be considered as separate. The first descriptor in the chain is pointed to by the field F(ELT). The chain is linked via the UI field of each descriptor in the order the linkage addresses were assigned. The linkage address is stored in the LKADR field. The output record is generated as a scatter-load word, containing the segment indicator and the address of the first entry, followed by the table of two-word entries. The two-word entry (Figure 14(c)), contains a \$LINK\$ subprogram (JUMPWORD) in the first word, while the second word contains the segment number in which the defined symbol is located and the absolute address assigned to the defined symbol. The segment number corresponds to the segment table described above. Each entry in the chain is processed until the output buffer fills up or until the chain is exhausted. If the output

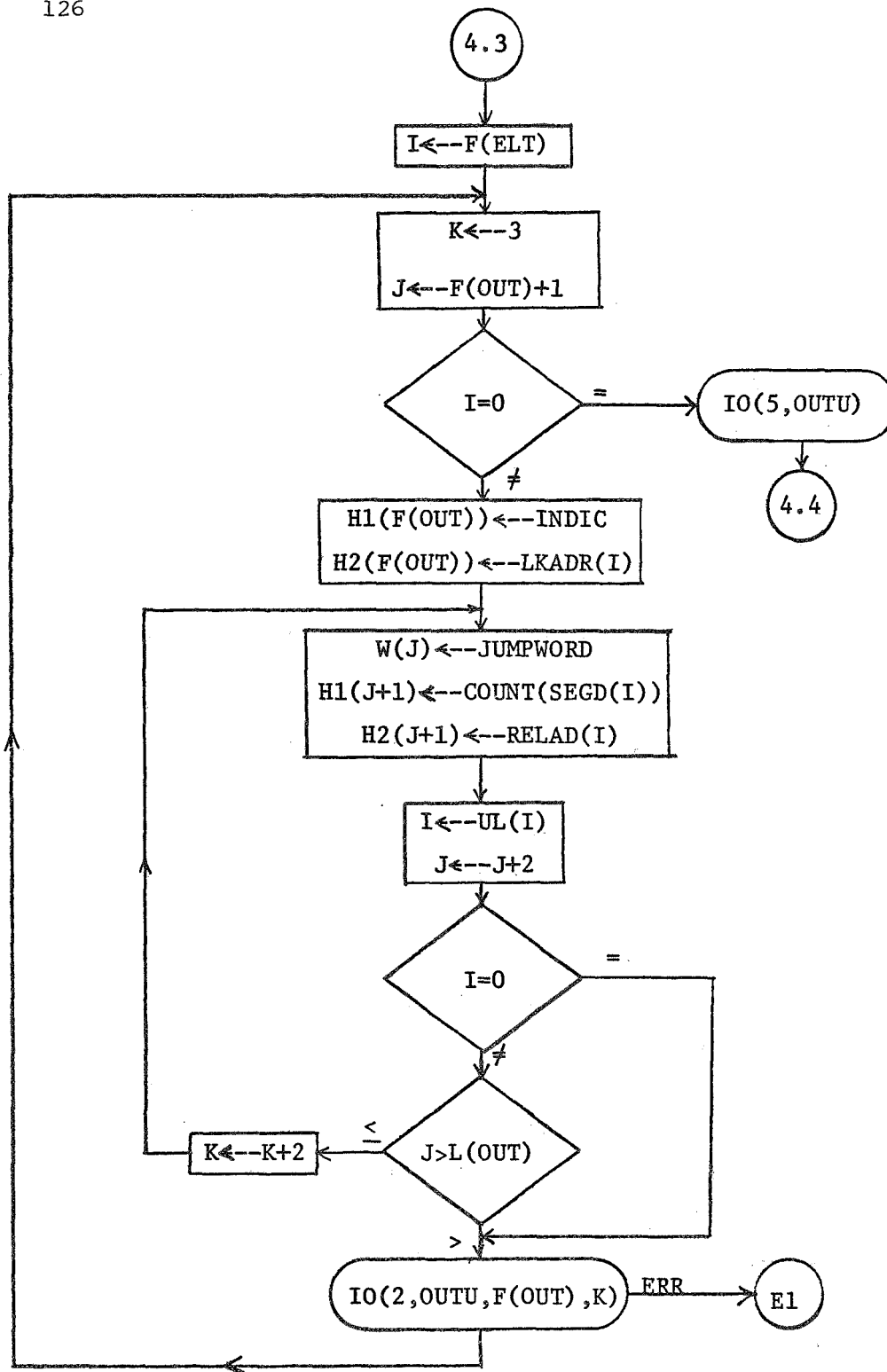


Figure 59

buffer fills up, it is output and processing continues. When the chain is exhausted, the output buffer is output. The absolute element is now complete, and an end-of-file mark is written on the output unit CUTU.

The last step, the print-out, is omitted from the report as it depends to a large extent on formatting of symbolic output, a procedure that varies from machine to machine.

At this point, the relocatable code translation function is described. The function performs four separate tasks:

- (a) each relocatable word is unpacked from the input buffer, translated, and packed into the output buffer;
- (b) if the relocatable word was an address referencing a global symbol, it checks to see if the reference was to another segment;
- (c) if the output buffer fills up, it outputs the buffer and re-initializes;
- (d) if the end of an input buffer is reached and another block of relocatable code remains on mass storage, then the block is input.

The four tasks are described in the form of flow charts in Figures 60 and 61. Tasks (a), (c), and (d) are described in Figure 60, while task (b) is described in Figure 61.

The second task of the translation routine is to recognize all inter-segment addressing. This is provided with the function CHECKLINK shown in Figure 61. This function checks to see if the defined symbol at location X is in the same segment as the subprogram represented by the element descriptor at location I. If so, then T is set to zero and the function returns to the translation. If not, T is set to one, and a check is made to see that a linkage entry is allocated to the defined symbol. The linkage entry

is provided by storing the address of a two-word cell from the available memory space into the IKADR field of the defined symbol entry and linking the defined symbol entry to a chain of linkage entries. Also, a check is made to ascertain whether the address reference is between segments that overlay each other. If so, depending upon the value assigned to the input variable OK, an error termination occurs, or the translation continues.

The translation requires unpacking of the relocatable code from the input buffer and repacking the executable code into the output buffer. The relocatable input and executable output is in the form of a string of relocatable words, each of varying lengths. In order to facilitate description, two string pointers, BI and BO, are used as pointers to the current byte in the strings located in the input and output buffers, counting from the beginning of the buffers. The function BYT(i,j,k) is used to indicate the k bytes beginning with the j-th byte counting from memory word i. Other variables used are NBYT for the number of bytes in a memory word, KT for the number of words currently stored in the output buffer, TKT for the total number of words of executable code produced for the subprogram, and SW for the switch that indicates that no more input remains.

The first, third, and fourth tasks are shown in Figure 60. These tasks are initialized by setting the variables to their initial values. Each time the output buffer is initialized the first word of the buffer is loaded with a scatter load word. The segment indicator INDIC is stored in the first half of this word, and the address at which to load the block of executable code is stored in the second half. This address is equal to the subprogram address ABSICC plus the total number of words of executable code already produced for the subprogram, TKT. At this point, the translation of the relocatable code begins. The

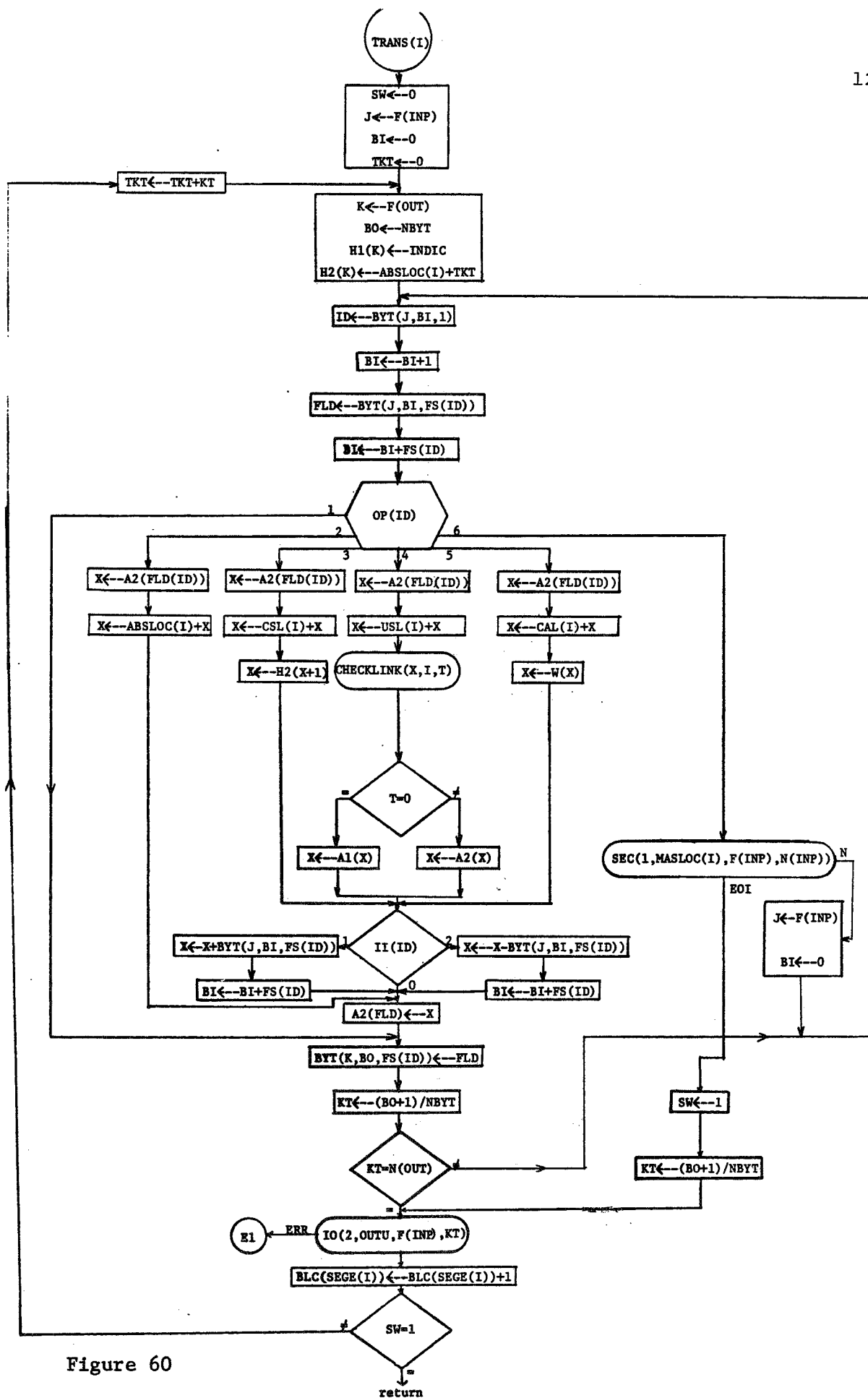


Figure 60

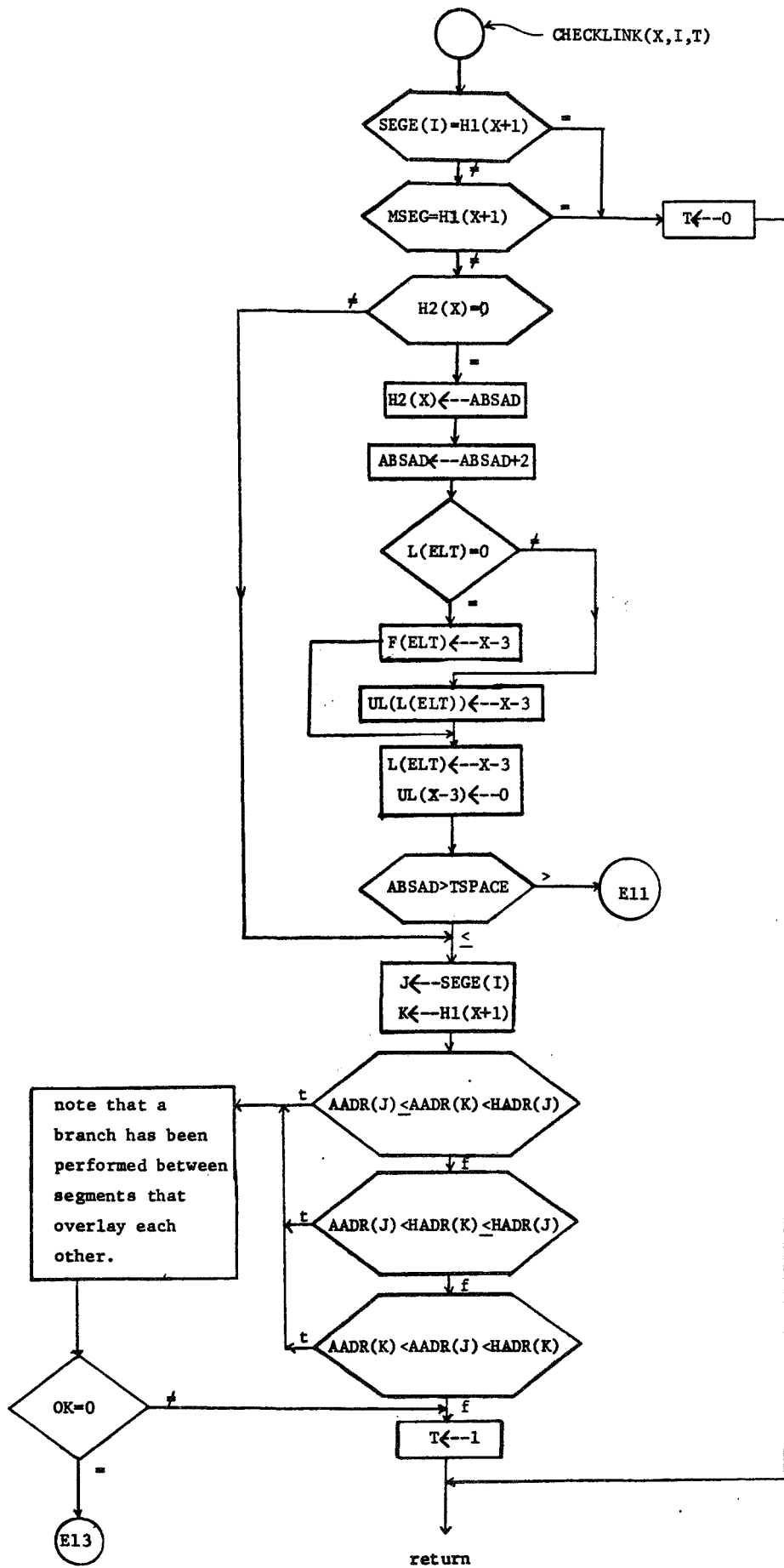


Figure 61

variable ID is loaded with the first byte of the relocatable word (ID←EYT(J,EI,1)). There are three fields in this byte: CP(ID) is the operator field, FS(ID) is the size of the field to be translated, II(ID) is the increment indicator field. Next, the FS(ID) bytes of the field to be translated, whether data, address, or table index, is unpacked into FID.

The translation of the relocatable word is a function of the contents of the field CP(ID). If CP(ID) is equal to 1, the contents of FID are data and no translation is necessary. If CP(ID) is equal to 2, the contents of FID represent an address relative to the subprogram and are added to the subprogram address. If CP(ID) is equal to 3, FID contains an index to the common symbol list for a common area address, and this address must be retrieved from the CABSIC field of the common area descriptor indicated by the address in the common symbol list. If CP(ID) is equal to 4, FID contains an index to the undefined symbol list, and depending upon the result of calling the function CHECKLINK, either the absolute address of the defined symbol or the absolute address of the linkage entry for that defined symbol is retrieved. If CP(ID) is equal to 5, then FID contains an index to the complex address table, and the contents of that table entry must be retrieved. If CP(ID) is equal to 6, then the end of the relocatable input has been reached and an attempt to make by the secretary to input another block of relocatable code.

In the cases of CP(ID) equal to 3, 4, or 5, then the possibility of an increment to the address indicated by the relocatable word exists. If II(ID) is equal to 0, there is no increment. If II(ID) is equal to 1, then the contents of the next FS(ID) bytes in the input string are added to the address stored in X. If II(ID) is equal to 2, then the contents of the next FS(ID) bytes in the input string are



subtracted from the address stored in X.

Once the address is complete, it is placed into the address portion of FLD. In all cases, data or address, the contents of FLD are placed into the next FS(ID) bytes of the output buffer, and translation proceeds with the next relocatable word.

If OF(ID) is equal to 6, another block of relocatable code is input and translation continues. However, if an end-of-file mark is reached, the contents of the output buffer are output, and relocatable translation of the subprogram is complete. The output buffer is output either at the end of the translation or whenever the output buffer fills up ( $KT=N(CUT)$ ).

Figure 15 shows the absolute element produced from the example. The first block is the header record, the second block is the executable code for subprogram MAIN, the third block represents the executable code for subprogram SUBRTN, and the fourth block represents the executable code for the subprogram SINCS. The second block results from the translating the relocatable code in the \$TEXT control record of subprogram MAIN shown in Figures 5(a) and 5(b). The subprogram descriptor for MAIN is shown in Figure 54 at locations  $1005_8$  through  $1020_8$ . The common symbol list begins at  $1020_8$ , the undefined symbol list begins at  $1015_8$ , and the subprogram address is found in the ABSLOC field ( $1012_8$ ).

The first word of the second block of the absolute element contains the main segment indicator ( $555555_8$ ) in the first half, and the subprogram address in the second half ( $1012_8$ ). Relocatable words 1 and 2 represent data and have translated directly to words 1 and 2 of the executable code. Relocatable words 3, 5, 7, 9, 11, 13 and 15 represent non-address parts of instructions and have translated directly

to the first halves of words 3, 4, 5, 6, 7, 8, and 9 of the executable code. Relocatable words 6, 8, 12, 14, and 16 represent addresses relative to the subprogram. These relocatable words translate, by adding them to 1012<sub>g</sub>, to the second halves of words 4, 5, 7, 8, and 9 in the executable code. The relocatable word 4 is a reference to an external address. The contents of the address portion of the FLD field index the second location in the common symbol list. This is the entry at 1016<sub>g</sub> in the tables which points to the absolute address in the first address portion of the word 1031<sub>g</sub>. The address in this location is 1224<sub>g</sub>. This address is placed into the address part of FLD and the whole field becomes the second half of word 3 in the executable code. Similarly, the external addresses specified by relocatable words 10 and 18 become the second halves of words 6 and 10 in the executable code. Relocatable word 17 specifies a common area. The FLD field indexes the first location in the common symbol list for this subprogram (located at 1020<sub>g</sub>). This contains the address of the common area descriptor for BICCR, which has been assigned absolute address 1000<sub>g</sub>. The increment indication field of the relocatable word is equal to 1, and so the contents of the increment field (5) is added to the address (1000<sub>g</sub>) to produce the address stored in the first half of word 10 in the executable code. It should be noted that in all address modifications, the translation only effects the address part of the FLD field, and the contents of the rest of the field are left untouched.

### 5.0 The Loader In Terms Of A Unified Hardware-software Design

The contention of this paper is that through close examination of system software, a hardware-software mix can be achieved that will improve the performance of the system. This section examines the algorithm for the loader (presented in section 4) in general and a particular function of the loader in detail in order to provide evidence to this contention.

In order to draw reasonable conclusions, a measure of performance is defined. In establishing this measure, we will assume that the overall software design of the system is fixed. This removes discussions of the merits of different algorithms from the scope of the paper. Within this framework, an improvement in performance of a system program will be defined as a reduction in the system's resources allocated to that program (i.e., systems overhead). Measurement of systems resources can be separated into the specific demands placed upon those resources: processor time, memory, and auxiliary memory. Within this paper we will concentrate on two: central processor time and memory. Central processor time (CPU time) will be measured in terms of main memory cycles. Memory usage will be represented by the space-time integral

$$\int_{t_1}^{t_2} M(t) dt$$

where  $t_2 - t_1$  represents the period that memory was used and  $M(t)$  represents the memory used at time  $t$ . In a word addressable memory, the units are words/second. The measurement of CPU is a traditional method while the memory measurement has only become necessary since the advent of

multiprogramming. As an example of the magnitude of systems overhead in contemporary systems, the UNIVAC 1108 EXEC 8 System requires a minimum of 65000 words of memory, continually.

By itself, the definition of performance improvement is not satisfactory because it does not include the concomitant change in monetary cost. Although a 10 per cent reduction in CPU time and memory word/seconds may be achieved, if the cost of the computer doubles then the overall benefit is debatable. Therefore, any hardware refinements must take into consideration the cost.

Given a method of measuring performance it is helpful to examine previous hardware refinements that have resulted in a reduction of system overhead. Among these are the introduction of index registers and multiple accumulators, and the development of hardware indirect addressing sequences. These changes and others have led to larger and more powerful instruction sets. Index registers and indirect addressing allow intricate addressing to be performed by a single instruction, resulting in a saving in CPU time and memory. Part of the saving in CPU time is due to the reduction of instruction fetches from main memory, and part is due to the fact that more is being done in the instruction fetch cycle. The addition of more hardware registers is advantageous principally because it provides a high speed local store for the storage of frequently used variables and storage of partial results. Another noteworthy advance has been the advent of partial word addressing. This allows for the efficient packing of information (freeing memory) and retrieval of information (freeing memory cycles). One of the more pertinent additions has been the inclusion of special instructions that are aimed particularly at alleviating system overhead. For example, the UNIVAC 1108 instruction set includes a set

of instructions for searching linear lists in contiguous memory locations. These instructions provide the power of a subroutine in one instruction.

It may be possible to generalize these past successes as follows:

- (a) judicious selection of hardware often precludes awkward and repetitive instruction sequences;
- (b) often-repeated instruction sets should be isolated and the possibility of providing a hardware replacement considered (in particular, list operations should be examined in this light);
- (c) the cost of instruction fetch should not be overlooked.

The generalizations above are by no means a complete set, but they do provide some direction to the study of a better hardware-software mix.

At this point it is worthwhile to discuss the method of study. Hardware implementation and testing is extremely expensive in the traditional manner. It is better if a simulation of the proposed hardware implementation can be effected first in software. Simulations are generally slow but allow extensive and inexpensive debugging to be carried out.

Later in this section (5.3), an implementation is described in detail in the Computer Design Language [16]. This language allows the user to describe the hardware configuration and logic in clear, precise statements. The simulator for this language [17] provides a method of testing the hardware description under a variety of input conditions.

## 5.1 Microprogramming As A Method Of Unified Hardware-Software Design

The problem with hardware implementations of traditional software functions is that, in general, the expense is only warranted in those functions that are used frequently. A second problem arises for complex hardware implementations if a design error is discovered after production has begun. A more basic problem is the fact that the cost varies in proportion with the complexity of the implementation. These three factors generally combine to cause the hardware designer to avoid hardware implementations of complex functions, functions that receive moderate use, or those functions that have any chance of undergoing a change in design.

These problems have recently been circumvented by the microprogrammed (or stored logic) computer, which combines the speed and parallelism of hardware with the variety and ease of modification of software. A microprogrammed computer can be described as a computer within a computer, a description that stresses its dependence on two memories. In the original thesis advanced by M. V. Wilkes [18], conventional machine instructions can be viewed as a number of sequential or parallel register-to-register transfers carried out under control of a small "microprogram."

This is precisely the form a microprogrammed computer takes. There are two memories: the traditional main memory and a smaller, faster control memory. The control memory may be read-only. The hard-wired instruction set consists of a set of micro-orders indicating simple register operations, main memory read and write, and control memory read and, possibly, write. A formal set of machine instructions is implemented by providing a microprogram that

interprets and carries out the functions specified by each instruction word.

Some of the possibilities are immediately evident. One microprogrammed computer could be programmed to execute the instruction sets of many computers. This is, in fact, the primary use of microprogramming today. The IBM 360 series machines offer upward compatibility to IBM 1401 and IBM 7090/7094 programs and inter-machine compatibility for 360 programs [19]. This process is called emulation. Recent proposals along these lines have been to microprogram computers to directly execute a higher-level language such as FORTRAN [20]. One such implementation, for language FULF, has been effected [21]. Many other uses of microprogramming have been proposed [22] [23] [24] [25].

Whether or not the microprogrammed computer is to be viewed as the next step in the evolution of a computer, a position held by many [26] [27], it obviously provides a convenient method of testing new hardware logic. Some of the advantages:

- (a) modification of the logic is a simple task of re-programming.
- (b) conversion to hardware is straight-forward once the logic is debugged,
- (c) complex functions can be implemented as easily and at little more cost than simple functions,
- (d) specific application functions may be implemented in microprogram and only loaded into control memory when they are to be used.

## 5.2 Loader Functions Amenable To Microprogram

This section proposes the implementation of a number of loader functions by microprogram in order to improve the performance of the loader. The examples were chosen because they are representative of the functions that are performed often within other system programs or because they show the range of functions that can be implemented in this fashion. The examples discussed are:

- (a) implementation of the queue in hardware,
- (b) locating the table pointers and hash-table in control memory,
- (c) implementation of search instructions for linked lists,
- (d) microprogramming the complex calculations,
- (e) microprogramming the translation of relocatable code to executable code.

The last example is described in detail in a separate subsection.

The queue described in the algorithm is used to store the mass storage locations of those subprograms that have been located in the library index. A queue has entries added at one end and retrieved from the other. The proposal is to have the first several locations and the last several locations of the queue located in a fixed set of hardware registers with the rest residing in main memory. A microprogram would automatically transfer the information from register to register, from main memory to register, and from register to main memory as entries were deleted from or added to the queue. This microprogram would also maintain the list of available space from which the queue received its entries. This concept has been implemented on the Burroughs B5500 [ 1] in the form of a push-down stack. The



advantages besides the automation of list maintenance, is that all access to the information is at register-to-register speeds. This implementation would be extremely useful to any system program using stacks, queues, or doubly-linked lists.

The table descriptors described in Figure 18 are used with high frequency throughout the algorithm. If they were located in a high speed local store (register memory), access times for retrieving the data in the respective tables could be halved. This is also true of the small hash table used within the defined symbol table (Figure 25). In fact, loading any of the small tables that are used within the algorithm would be advantageous. It should be noted that these tables need only be loaded when they are to be used.

The previous discussion of the UNIVAC search instructions suggests a more ambitious set of list operations. Systems programs use linked lists for most data storage (singly-linked, doubly-linked, tree structures, circular lists, etc.). The flow charts in Figures 34, 36, 39, 45, 46, and 51 all contain list searching sequences. A set of primitive search operations would be extremely useful. These could be microprogrammed to respond to list operation instructions that would specify the address field (or fields) to search along, the field of the list cell to be searched, and several values to respond to (the value desired, the end of list indicator, etc.).

The flow charts in Figures 31 and 32 show the complex calculation function that performs the operations indicated by instructions of the form shown in Figure 6. A set of complex calculation instructions is associated with each subprogram. These instructions reference data via one of the three symbol lists (undefined, common, and complex) for the subprogram. By creating a microprogram to interpret the

instructions and perform the indicated operations, and locating the three symbol lists in a register store, a savings in time and memory can be achieved. This is not difficult to understand because the comparison is between implementing the interpreter in slow memory versus implementing the interpreter in fast memory.

### 5.3 Microprogramming the Translation of Relocatable Code To Executable Code

The last example is the microprogramming of the function that performs the relocatable code translation of the loader. The conversion to microprogram requires three steps, each involving some change in the algorithm itself. First, the algorithm is flow charted from the existing software. At this point, it may be desirable to substitute a more clear description of the algorithm then to translate the software code to flow chart exactly. In our case, this step is represented by the flow chart in Figure 60. The second step is to convert the flow chart to hardware sequence charts. At this point decisions about registers and memory must be made. The algorithm itself may be modified to take advantage of a particular hardware implementation. For example, the hardware addition of a shift register obviates the need for the complicated BYT function used in the flow chart. The third step is the conversion from sequence charts to microprogramming. At this point, the hardware control is defined and an attempt is made to minimize the execution time. The minimization is accomplished by maximizing parallel operations and by performing as many tasks as possible in a memory cycle.

This example is a modification of an example presented in a previous report (28). The reader is directed to that report for the details of the implementation. In order to simplify the presentation, only part of the flow chart in Figure 60 was implemented. The complex address operator (OP=5) was deleted, and the value of 5 was used to indicate the end of translation. The microprogram assumed that the input buffer was already loaded and terminated when OP was equal to 5 rather than attempt another input. The function CHECKLINK was ignored and the RELAD field is assumed to be switched with the LKADR field in the defined symbol description word (T#0 in the flow chart).

The design of the configuration assumes a word format like that of an IBM 7090/7094. The configuration, micro-orders, and microprogram are described by the Computer Design Language (CDL). This language, and the methodology above, are attributable to Y. Chu (16) (28).

Implementation of the relocatable translation function as a microprogram supports the arguments for the use of microprogramming in a unified hardware-software design. As shown below, reductions in CPU time and memory requirements are effected.

#### Savings in Computer Time

It is possible to estimate the speed of translating the executable code (i.e., instructions or data words). At one extreme, it requires 5 main memory cycles to translate a two-address instruction where the addresses are external addresses. (It requires two

relocatable words for one such instruction.) Let the main memory cycle time be 1 microsecond. Then, the microprogram controlled translator is capable of producing from 77,000 to 200,000 instructions or data words per second. The required translation time for a two-address instruction could be further lowered to 9 main memory cycles if the symbol lists for the relocatable code resided in control memory.

For the sake of comparison, the software implementation of the relocatable translation on the IBM 7090/7094 Loader (IFLDR) was examined by counting lines of code and charging one memory cycle for instruction fetch and execution (assuming data fetch is overlapped), it was found that the microprogrammed translator reduces CPU time by a factor of 10. A fairer comparison may be to implement the algorithm in Figure 60 in software on a machine with a larger instruction set (e.g., that of the UNIVAC 1108). In this case, the microprogrammed translator still represents an improvement, reducing CPU time by a factor of 3.

#### Core Savings

Similar analysis reveals that a software implementation requires between 100 to 200 memory locations to hold the machine instructions. The microprogrammed translator frees these locations for other use. This particular gain is somewhat dampened by the requirement made upon control memory by the microprogram. However, it should be noted that the microprogram requires only 24 words of storage rather than 100 or more. The variability and parallelism of the micro-instruction make this possible.

#### Saving In Hardware

As the cost of hardware fabrication can override the benefits of reduction in CPU time and main memory usage, it is worthwhile to compare the hardware described with that of existing computers. The IBM 7090/7094 provides a good example as it has been used as the basis for the comparison.

The hardware required for the algorithm, shown in Figure 62, closely approximates the register set on the ILM 7094 (e.g., seven index registers, an accumulator, a multiplier/quotient register, an instruction register, etc.). Therefore, it can be said that the proposed internal register structure of the microprogrammed machine approximates that of conventional computers. The major difference is, of course, the control memory. It should be noted that the cost of a control memory is offset by the savings achieved by dispensing with wired-in logic.

## 6.0 Summary, Conclusions, And Recommendations

This paper set out to demonstrate the advantages of a unified hardware-software design approach to an operating system through microprogramming by concentrating on a single system program, the loader. The loader was considered in its broader sense of a relocatable allocator and linkage editor. The functions of the loader were reviewed, a specific description of the input and output given, and a detailed algorithm set forth. Microprogramming was advanced as a method of achieving hardware-software integration. The algorithm was examined for functions that would benefit system performance if implemented by microprogram. Several such functions were outlined and one was described in great detail and implemented through simulation.

By reviewing the loader in a general setting and in some detail, many design problems have been presented. In the discussion of overlay segmenting, the comparison of tree structured segmentation with unstructured segmentation has shown the advantages and disadvantages of both. The tree structure allows many decisions to be made about locating subprogram and common areas, but is restrictive. The unstructured method allows more freedom in segmentation and simplifies automatic loading.

An implicit result in the study of the loader is the necessity of auxiliary storage to the implementation of the algorithm. The need arises because, in the upper limit, the sum of the relocatable elements must exceed that of the finished program (even with packing of information). Furthermore, the use of auxiliary storage has a direct effect on the use of data structures. For example, the algorithm presented in section 4 stored the relocatable code in auxiliary storage. An algorithm that relied upon

chaining of addresses within the blocks of relocatable code would have been more difficult to implement.

In addition, the study of the loader allowed several types of data structures to be compared for ease of storage and retrieval and maintenance cost. For example, hash-coding was compared with linear lists and binary tree structures. In the final design, hash-coding was combined with a simple list structure. This reduced the requirements on memory for the hash table. The binary tree structure was discarded because of the cost of rebalancing the tree.

The advantages of the unified hardware-software design were re-enforced by enumerating past successes, outlining some possibilities suggested by the loader, and presenting a method of future exploration. The approach was seen to reduce system overhead, CPU time and memory usage. Due to the possibility of high hardware costs, it was pointed out that a careful balance must be maintained.

The success of the implementation in section 5 underlines the advantages of microprogramming as a method of hardware-software design. Interpreters of specialized languages, in this case the relocatable language, can be implemented easily by microprogram. Therefore, rather than writing system programs in one general-purpose machine language, specific, high powered instruction sets can be developed for each function. Control memory could hold those interpretive microprograms used most often. In the loader, it may be advantageous to use a list processing language, a simple calculation language (as shown in Figure 6), and a language for relocation where each would each be interpreted by a separate microprogram in control memory.

In conclusion, it appears that the unified hardware-software design can be of immense benefit in the development of operating systems. Having examined the loader once, it

can be implemented in any minimal microprogrammed computer. This should mean a smaller lead time in designing a new system and a more reliable design. The direct benefit is a reduction in system overhead, while a by-product appears to be a more varied programming environment for all. However, in order to understand clearly the benefits of the unified hardware-software design approach, more ambitious studies should be undertaken, including:

- Study of the feasibility of microprogram controlled machines that directly execute higher-level procedural languages. A number of studies have proven the possibility of such machines.
- Studying in detail a microprogram controlled machine that must use a backing store to hold its set of microprograms. A study of this nature could draw heavily on work done in paging and segmenting systems. This study is necessary in order to determine when it is feasible to use backing store to store microprograms.
- Implementation of a language and simulator that would allow detailed specification and simulation of a complete microprogram controlled operating system. The language should be able to simulate asynchronous processes, rotating memory devices, and provide statistics upon the performance. This could be an extension of an existing language like the Computer Design Language.
- Study, design, and implementation of a complete system. Each phase should be attacked with the whole system in mind in order to provide conscientious debate on the validity of past



systems designs.

- Implement the dynamic loader as a microprogram.  
This could be extended to a general segmentation or paging system.

## APPENDIX

## Description Of Input In Backus-Naur Form

The input to the loader is summarized below in Backus-Naur Form (BNF). The Backus-Naur Form consists of a series of definitions. Definitions are productions of the form

$\langle \text{left-side} \rangle ::= \langle \text{right-side} \rangle,$

which is read, " $\langle \text{left-side} \rangle$  is defined as  $\langle \text{right-side} \rangle$ ."

The brackets  $\langle \rangle$  are used to enclose syntactic categories. The right-side of any production is one or more definitions made up of other syntactic categories or terminals. Alternative definitions are separated by the vertical slash ( $|$ ). Terminals are unbracketed characters or strings of characters. In order to leave the definition as general as possible, some syntactic categories are defined as a sort of higher level terminal. These syntactic categories are marked with an asterisk (\*). This is done in those cases where assumptions would have to be made about the machine specification (word formats, character set, memory size, etc.) in order to define the terminal string. For example, to define further the make-up of a  $\langle \text{segment name}^* \rangle$  requires knowledge of the allowable characters; to define further  $\langle \text{relative address}^* \rangle$  requires knowledge of the number of addressable locations. Rather than restrict the algorithm, further definition is left to the particular implementation.

The BNF is extended by adding references to the flow charts that perform the semantic interpretation. This is only done for the major categories and always refers to the left-side of the definition which it follows. These references are bracketed by the "bullet" marks ( $\bullet$ ). For example, the control words \$RELOC, \$SEG, \$COMM, \$UND, \$DEF, \$CMPLX, and \$TEXT are recognized in flow chart 1.2 (Figure 29). Therefore the definition of  $\langle \text{control word} \rangle$  in this

format is:

```

<control word> ::= $RELOC | $SEG | $CCMN |
                  $UND | $DEF | $CMPLX |
                  $TEXT           □ 1.2 □

```

This form is used only for the major syntactic categories. The syntactic categories that comprise the definition of these major categories are either interpreted within the same flow chart or stored for later interpretation.

```

<loader input> ::= <loader command><program>
                  <end of information>           □ 1.0 □

```

```

<loader command> ::= $ABSLT

```

```

<program> ::= <unsegmented program>|
              <segmented program>

```

```

<segmented program> ::=
    <segment descriptor><subprogram set>|
    <segmented program><segment descriptor>
    <subprogram set>

```

```

<segment descriptor> ::= <segment delimiter>
                        <segment name*><relation part> □ 1.3 □

```

```

<segment delimiter> ::= $SEG

```

```

<relation part> ::= <null>| ( <predecessor list> )

```

```

<predecessor list> ::= <predecessor>|
                      <predecessor list>,<predecessor>

```

<predecessor> ::= <segment name\*>

<unsegmented program> ::= <subprogram set>

<subprogram set> ::= <relocatable element> |  
                   <subprogram set><relocatable element>

<end of information> ::= <EOI mark\*>     □ 2.0 □

<relocatable element> ::= <preface><preamble><text>

<preface> ::= <preface delimiter><subprogram name\*>  
                   <block indicator\*><subprogram size\*>  
                   □ 1.4 □

<preface delimiter> ::= \$RELOC

<preamble> ::= <common symbols><undefined symbols>  
                   <defined symbols><complex symbols>

<common symbols> ::= <null> |  
                   <common delimiter><number of entries\*>  
                   <common symbols table>   □ 1.6 □

<common delimiter> ::= \$COMN

<common symbols table> ::= <common table entry> |  
                   <common symbols table><common table entry>

<common table entry> ::= <symbolic name\*>  
                   <block size\*><relative address\*>

<undefined symbols> ::= <null> |  
                   <undefined delimiter><number of entries\*>  
                   <undefined symbols table> □ 1.5

```
<undefined delimiter> ::= $UND
```

```
<undefined symbols table> ::=
    <undefined table entry>|
    <undefined symbols table>
        <undefined table entry>
```

```
<undefined table entry> ::= <symbolic name*>
```

```
<defined symbols> ::= <null>|
    <defined delimiter><number of entries*>
        <defined symbols table> □ 1.8 □
```

```
<defined delimiter> ::= $DEF
```

```
<defined symbols table> ::= <defined table entry>|
    <defined symbols table><defined table entry>
```

```
<defined table entry> ::=
    <symbolic name*><value*><relative address*>
```

```
<complex symbols> ::= <null>|
    <ccomplex delimiter><number of entries*>
        <table size*><complex table> □ 1.7 □
```

```
<ccomplex delimiter> ::= $CMPLX
```

```
<ccomplex table> ::= <complex table entry>|
    <complex table><complex table entry>
        □ 3.1 □
```

```
<complex table entry> ::= <lcard operator*>
    <arithmetic expression><store operator*>
        <table index*>
```

```

<arithmetic expression> ::= <table item>|
    <arithmetic expression>
    <arithmetic operation><table item>

<table item> ::= <table indicator*><table index*>

<arithmetic operation> ::= <addition>|<subtraction>|
    <multiplication>|<division>

<text> ::= <text delimiter><text body><text end*>
    □ 1.9 □

<text delimiter> ::= $TEXT

<text body> ::= <text word>|<text body><text word>
    □ TRANS □

<text word> ::= <data word>|<address word>

<data word> ::= <data operator*><field size*>
    <data item*>

<address word> ::= <relative address>|
    <common address>|
    <undefined address>|
    <complex address>

<relative address> ::= <relative address operator*>
    <field size*>relative address*>

<common address> ::= <common address operator*>
    <field size*><address part>

<external address> ::= <external address operator*>

```

<field size\*><address part>

<ccomplex address> ::= <ccomplex address operator\*>  
 <field size\*><address part>

<address part> ::= <no increment indicator\*>  
 <table index\*>|  
 <add increment indicator\*><table index\*>  
 <increment\*>|  
 <subtract increment indicator\*>  
 <table index\*><increment\*>

## BIBLIOGRAPHY

## Abbreviations:

CACM - Communications Of The ACM

IEEEIC - IEEE Computer Transactions

- ( 1 ) A Narrative Description Of The Burroughs B5500 Master Control Program. Detroit, Michigan: Burroughs Corporation, 1969.
- ( 2 ) Daley, Robert C. And Dennis, Jack E., "Virtual Memory, Processes, And Sharing In MULTICS," CACM, May, 1968, pp. 306-312.
- ( 3 ) Dennis, J. E., "Segmentation And The Design Of Multiprogrammed Computer Systems," in Programming Systems And Languages, Edited by Saul Rosen (New York: McGraw-Hill Book Company, 1967), pp. 699-713.
- ( 4 ) IBM 7090/7094 IBSYS Operating System: Version 13. IBCPE Processor. IBM Systems Reference Library, File No. 7090-27 (Form C28-6389-2).
- ( 5 ) Linkage Editor, IBM System 360 Operating System. IBM System Reference Library, File No. S360-31-48 (Form C28-6538-1).
- ( 6 ) Lanzano, E. C. "Loader Standardization For Overlay Programs." CACM, October, 1969, pp. 541-550.
- ( 7 ) Collector, UNIVAC 1108 Operating System Technical Documentation. Volume 8. PCR# 11. UNIVAC



Division Of Sperry Rand Corporation, 1968.

- ( 8) IBM 7094 Principles Of Operation. IBM Systems Reference Library, File No. 7094-01 (Form A22-6703-4).
- ( 9) Pankhurst, R. J. "Program Overlay Techniques." CACM, February, 1968, pp. 119-125.
- (10) Flores, Ivan. Computer Software. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1965.
- (11) McCarthy, J.; Corbato, F. J.; and Daggett, M. M., "The Linking Segment Subprogram Language And Linking Loader," in Programming Systems And Languages, Edited by Saul Rosen (New York: McGraw-Hill Book Company, 1967), pp. 572-581.
- (12) Maurer, W. D. Programming: An Introduction To Computer Languages And Techniques. San Francisco: Holden-Day, Inc., 1968.
- (13) Wegner, Peter. Programming Languages, Information Structures, And Machine Organization. New York: McGraw-Hill Book Company, 1968.
- (14) Knuth, D. E. The Art Of Computer Programming: Volume 1, Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Company, 1968.
- (15) Morris, Robert, "Scatter Storage Techniques," CACM, January, 1968, pp.38-43.
- (16) Chu, Y. "An Algol-like Computer Design Language,"

CACM, October, 1965, pp. 607-615.

- (17) Mesztenyi, C. K., "Computer Design Language, Simulation And Boolean Translation," Technical Report 68-72, Computer Science Center, University Of Maryland, June, 1968.
- (18) Wilkes, M. V., "The Best Way To Design An Automatic Calculation Machine," Manchester University Computer Inaugural Conference, p. 16, 1951.
- (19) Tucker, S. G. "Emulation Of Large Systems," CACM, December, 1965, pp. 753-761.
- (20) Melbourne, A. J. And J. M. Pugmire. "A Small Computer For The Direct Processing Of FORTRAN Statements." Computer Journal, 8(1965), pp. 24-27.
- (21) Weber, Helmut. "A Microprogrammed Implementation Of EULER On IBM System/360 Model 30," CACM, September, 1967, pp. 549-558.
- (22) Hawryszkiewicz, Igor T. "Microprogrammed Control In Problem-Oriented Languages," IEEE TC, October, 1967, pp. 652-658.
- (23) Iversen, Kenneth E. A Programming Language. New York: John Wiley And Sons, Inc., 1962.
- (24) Lawson, Harold W., Jr. "Programming-languages-oriented Instruction Streams," IEEE TC, May, 1968, pp. 476-485.
- (25) Rosin, Robert F. "Contemporary Concepts Of

Microprogramming And Emulation," Computing Surveys, December, 1969, pp. 197-212.

- (26) Rakoczi, Iaszlo I. "Fourth Generation Computer System Product Line," Form 1007-1, Los Angeles, Calif.: Standard Computer Corporation, 1968.
  
- (27) Flynn, Michael J. and M. Donald MacLaren. "Microprogramming Revisited," Proceedings ACM 22nd National Meeting, 1967, pp. 457-464.
  
- (28) Chu, Yachan; Fardo, Cliver R.; And Yeh, Jefferey; "A Methodology For Unified Hardware-software Design." Technical Report 70-107, Computer Science Center, University Of Maryland, January, 1970.